From MetroII to Metronomy, Designing Contract-based Function-Architecture Co-simulation Framework for Timing Verification of Cyber-Physical Systems

Liangpeng Guo

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-11
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-11.html

March 13, 2015

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
13 MAR 2015		**00-00-2015 to 00-00-2015**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
From MetroII to Metronomy, Designing Contract-based Function-Architecture Co-simulation Framework for Timing Verification of Cyber-Physical Systems	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

As the design complexity of cyber-physical systems continues to grow, modeling the system at higher abstraction levels with formal models of computation is increasingly appealing since it enables early design veri cation and analysis. However, it is very challenging to analyze and verify timing at the early design stages, as the design representation is still abstract and trade-o s have to be made between the performance requirements de ned in terms of system functionality and the cost of the feasible architecture that can implement the functionality. In this work, we present Metronomy, a function-architecture co-simulation framework that integrates functional modeling from Ptolemy and architectural modeling from the MetroII environment via a mapping interface. Metronomy completely separates the function and architecture modeling. It allows the function and the architecture of the system to be modeled in the most suitable design environments. At the same time, Metronomy allows designers to do timing veri cation and design space exploration at early design stage by exploiting contract theory and co-simulation. Two case studies on an electrical power system and a paper-feed sub-system for a high speed printing press demonstrate the e ectiveness of our approach.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**64**	

From MetroII to Metronomy, Designing Contract-based Function-Architecture Co-simulation Framework for Timing Verification of Cyber-Physical Systems

by

Liangpeng Guo

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Alberto L. Sangiovanni-Vincentelli, Chair
Professor Edward A. Lee
Professor Lee W. Schruben

Spring 2015

From MetroII to Metronomy, Designing Contract-based Function-Architecture Co-simulation Framework for Timing Verification of Cyber-Physical Systems

Abstract

From MetroII to Metronomy, Designing Contract-based Function-Architecture Co-simulation Framework for Timing Verification of Cyber-Physical Systems

by

Liangpeng Guo

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto L. Sangiovanni-Vincentelli, Chair

As the design complexity of cyber-physical systems continues to grow, modeling the system at higher abstraction levels with formal models of computation is increasingly appealing since it enables early design verification and analysis. However, it is very challenging to analyze and verify timing at the early design stages, as the design representation is still abstract and trade-offs have to be made between the performance requirements defined in terms of system functionality and the cost of the feasible architecture that can implement the functionality. In this work, we present Metronomy, a function-architecture co-simulation framework that integrates functional modeling from Ptolemy and architectural modeling from the MetroII environment via a mapping interface. Metronomy completely separates the function and architecture modeling. It allows the function and the architecture of the system to be modeled in the most suitable design environments. At the same time, Metronomy allows designers to do timing verification and design space exploration at early design stage by exploiting contract theory and co-simulation. Two case studies on an electrical power system and a paper-feed sub-system for a high speed printing press demonstrate the effectiveness of our approach.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank my advisor Alberto Sangiovanni-Vincentelli. Without his mentorship, advice and support, I could have never been able to complete my PhD student career. He is not only an incredibly knowledgeable mentor but also a great resource of new ideas and motivations. Many of the ideas in this dissertation originated from our discussion. In addition to that, I will never forget how I was amazed by his enthusiasm and energy from time to time. As I go forward in my career, I will forever benefit from our interaction.

I would like to thank Prof. Edward Lee and Prof. Lee Schruben for the participation in both my qualifying exam as well as the thesis process. I also would like to thank Prof. Jan Rabaey for being on my qualifying exam committee. Their acute comments from various angles helped shape this research. As a UC Berkeley graduate student I have had the pleasure of working with some of the best professors in the world. I would like to thank Prof. Andreas Kuehlmann, Prof. Kurt Keutzer, Prof. Jaijeet Roychowdhury, Prof. Sanjit Seshia, Prof. George Necula, Prof. Edward Lee, and Prof. Lee Schruben for the excellent classes they taught.

I am also fortunate to interact with a group of talented colleagues and friends I have worked with over the past 6 years while at Berkeley. In particular Qi Zhu and Pierluigi Nuzzo, with whom I discussed details of this research work; Patricia Derler and John Eidson who provided the model of printing press in Ptolemy; Christopher Brooks who managed the Ptolemy software; Marco Di Natale, Haibo Zeng, Arkadeb Ghosal, Paolo Giusto, Alessandro Pinto, Alberto Puggelli, Alena Simalatsar, and Robert Passerone, with whom I collaborated on a number of research works. I would also like to extend my gratitude for the support and help from all the friends of the Donald O. Peterson (DOP) center. In particular current and past students from Alberto's group. They are Chung-Wei Lin, Baihong Jin, Antonio Iannopollo, John Finn, Nikunj Bajaj, Chen Lv, Guang Yang, Mark McKelvin, Mehdi Maasoumy, Xuening Sun, Kelvin Lwin, Guoqiang Wang, Douglas Densmore, Abhijit Davare, Claudio Pinello, and Mohammad Mozumdar. A special 'thanks' to Ruth Gjerde and Shirley Salanio for their excellent jobs in handling all the logistics. It is impossible to list everyone important to me here. Thank you for everything.

I have also received support from a couple of companies throughout the years as well. In particular General Motors (GM), United Technologies Research Center (UTRC) and National Instruments (NI), which have been open to my research and supported me during the internships. I would like to thank the management of these companies, especially Joseph D'Ambrosio from GM, Brian Murray from UTRC and Hugo Andrade from NI for supporting my research.

Last but not least, I am forever indebted to my parents Siping Guo and Faying Qu, who love me with all their hearts and always support me unconditionally. All the accomplishments in school would not have been possible without their support. I also would like to thank my girl friend Ying Zheng, with whom we spent all the most wonderful and difficult time of the last few years. And I am also grateful to my friends Guojun He, Di Zeng and Bohan Ye, who shared Apartment 5 with me. Hopefully we will one day realize what a unique and

enjoyable experience it was to study and live together at Berkeley pursuing our academic goals.

This work was partially supported by IBM and United Technologies Corporation via the iCyPhy consortium, and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. A part of this work has been published in [15].

Chapter 1

Introduction

1.1 Platform-based Design of Cyber-physical Systems (CPS)

Cyber-physical systems (CPS) are integrations of computation, networking, and physical processes. A modern cyber-physical system typically consists of sensors, actuators and software components running on distributed networked computational elements. The software/hardware components work together to monitor and control the physical processes. Designing a CPS is challenging due to the complex coupling of physical processes and computations. A slight change in the physical processes may affect the behaviors of the computational elements. And a slight change in the computation may affect the physical processes as well. With safety and reliability requirements, the coupling of physical processes and computations as well as the increasingly complex software/hardware architectures have made the design of complex cyber-physical systems ever challenging.

To address the challenges, the design of a modern cyber-physical system is often carried out by engineers from multiple domains, e.g. the control engineers and the software/hardware engineers. The control engineers focus on the interactions between the physical processes and computations and the software/hardware engineers focus on the correct implementations of the control algorithms. As the complexity of such systems continues to grow, it is highly beneficial to separate the design concerns of control engineers and software/hardware engineers and manage the complexity by a strict design methodology.

In the Platform-Based Design (PBD) methodology [32], two types of models are generally used to represent different design aspects: the *functional model* defines what the design does in terms of a set of services and the *architectural model* describes how these services are implemented by a collection of architectural primitives. Specifically, for CPS control design, the functional model is used to describe the control algorithm and its interaction with the physical plant, captured using formal models of computation (MoCs) for system verification and analysis. The architectural model is used to describe the implementation platform for the control algorithm, including embedded processors, sensors, actuators, communication

primitives, as well as the operating system, firmware and drivers.

The two types of models represent the different design concerns from the perspectives of control engineers and software/hardware engineers, who use different languages, methods and tools to represent, analyze and optimize the design. In CPS design, the system functionality is typically modeled in one environment (e.g. Simulink, Modelica) by control engineers, while the system architecture is modeled in a different one (e.g. SystemC or other programming languages) by software and hardware engineers. Since each design environment has its own strengths, it is almost impossible in practice to force control engineers and software/hardware engineers to use the same environment to represent, analyze and optimize the functional model and the architectural model.

However, although the functional model and the architectural models are separated, they rely on each other to achieve the system specification of a cyber-physical system. On the one hand, the correct functional behaviors rely on the timing of the implementation platform. On the other hand, non-functional properties of the implementation platform such as physical time, power consumption rely on the functional behaviors executed on the platform. In early design stages, the functional model is designed with explicit or implicit assumptions on the implementation platform and the architectural model is also designed with assumptions on the functional behaviors. For example, the sampling period of a block in the functional model imposes an implicit assumption that the implementation of the block should have an execution time that is less than the sampling period.

It is important to make sure the assumptions made by the functional model can be satisfied by the implementation platform and vice versa. In addition to that, the assumptions also lead to trade-offs between system performance and the cost of the implementation platform. As an example, a faster control loop can provide better performance but also requires a more expensive implementation platform; however, a slower control implementation tends to sacrifice performance in order to achieve a cheaper solution. To facilitate the exploration of such trade-offs, it is critical to analyze and verify the real-time performance of a system across the boundary between the functionality and the architecture. In fact, timing properties, such as sampling periods and latencies from sensing to actuation can significantly affect the control performance and even the functional correctness of the design. However, whether certain sampling periods are actually allowed and what the values of the sensor-to-actuator latencies are ultimately depend on the implementation platform.

This dissertation addresses these issues by formalizing the interactions between system function and architecture, and by providing a framework to efficiently co-simulate and co-analyze functional and architectural models. As a result, the design process benefits from the following features of the proposed framework Metronomy:

- *Complete separation of the functional model and the architectural model:* it allows users to model the functional model in Ptolemy and the architectural model in MetroII, which opens the possibility of representing, analyzing and optimizing the design aspects using different languages, methods and tools.

- *Formalized timing assumptions/guarantees between the functional model and the architectural model:* it exploits contract theory to formalize and organize the relationship of the functional model and the architectural model in terms of timing.
- *Co-simulation between the functional model and the architectural model:* it bridges the functional and architectural models with a co-simulation approach to allow analyzing the system properties that are relevant to both such aspects. Co-simulation of the functional model and the architectural model helps verify the functional behavior of the system. It also helps evaluate non-functional properties of the implementation platform such as physical time, power consumption, and monetary cost. Furthermore, function-architecture co-simulation helps the designers make assumptions that lead to better trade-offs between the system performance and the cost of the implementation platform.

1.2 Related Work

Metronomy utilizes the Ptolemy as the front-end for designing the functional model and MetroII as the back end for designing the architectural model. In this section, I will describe the main aspects of Ptolemy and MetroII, along with some other related design frameworks.

1.2.1 Ptolemy

Ptolemy II is a modeling and simulation environment for heterogeneous systems, which consists of several executable domains of computation that can be mixed in a hierarchy [7]. All models of computation are described operationally in terms of a common executable interface.

Specifically, a model in Ptolemy consists of a set of actors and a director. Actors are components that execute concurrently and share data with each other by sending messages via ports. The director specifies the Model of Computation (MoC) that determines the activation order of actors on the top level.

An actor can be *atomic* or *composite*. An atomic actor encapsulates basic computations, from simple arithmetic operations to more complex ones like an FFT [24]. The data can be shared between two atomic actors by sending messages via the ports and connections. Ptolemy has two types of composite actors. A *transparent composite actor* is a group of other actors. The enclosed actors are exposed to the governing director of the upper level as if the composite actor is transparent and the actors are placed on the upper level. An *opaque composite actor* is a composition of actors and a director. The enclosed actors are governed by the enclosed director, which is activated when the composite actor is activated on the upper level. In other words, the enclosed director and actors specify the behaviors of the composite actor when activated by the governing director on the upper level. The data can be shared between two levels by sending messages via the ports of the composite actor. Figure 1.1 from [30] shows a simple hierarchical Ptolemy model.

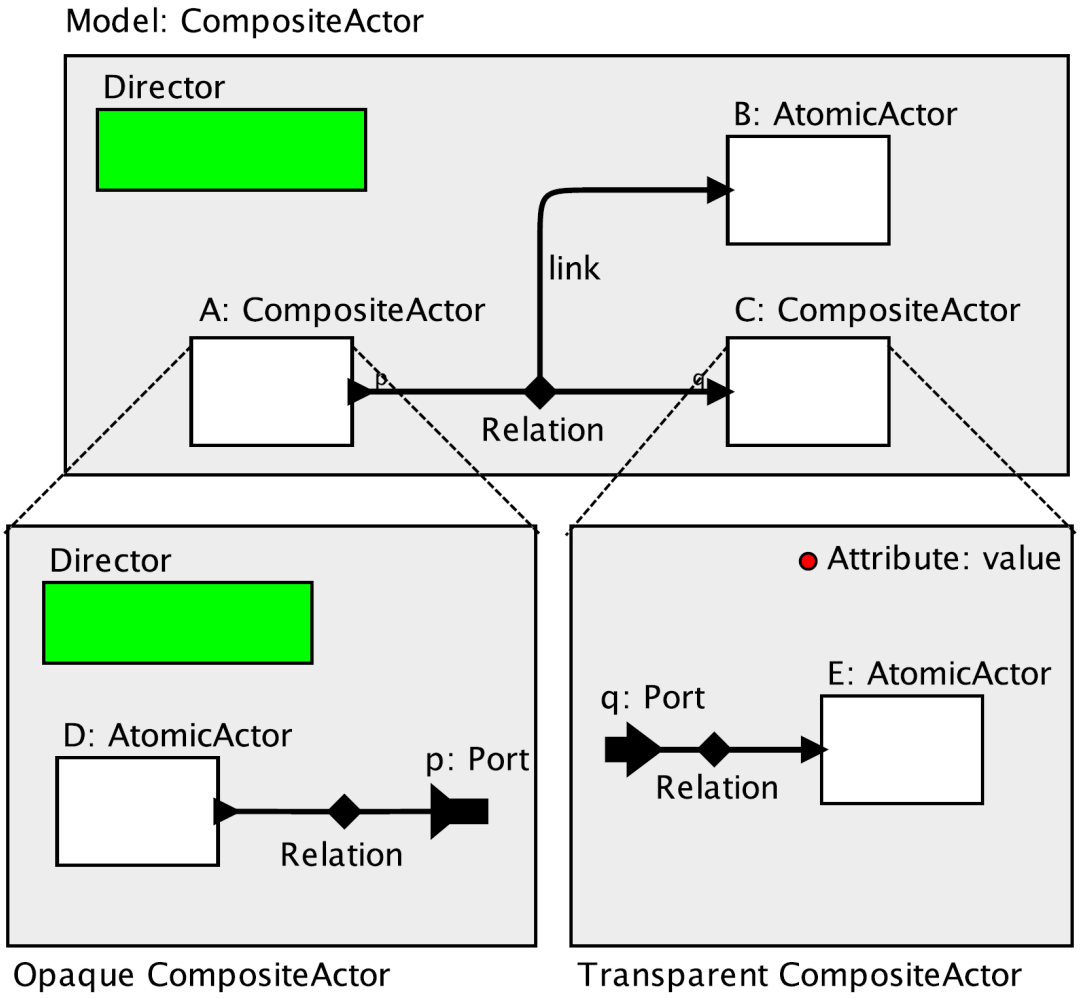

Figure 1.1: The hierarchical Ptolemy model from [30] consists of two composite actors and a director. The opaque actor contains a director and actors, which specify the behaviors on the secondary level. The transparent composite actor is simply a group of actors on the top level.

Ptolemy has a user-friendly GUI and features a number of commonly used MoCs, including heterogeneous modeling using continuous and discrete domains. But Ptolemy lacks support for the integration of high-fidelity models of implementation platforms, which are often conveniently built using domain-specific tools. In Metronomy, we bridge this gap by supporting co-simulation with implementation platform models developed in MetroII. To support such integration, we create a new co-simulation director *CoSimDirector* and extend the Ptolemy directors of the Discrete Event (DE), Synchronous Dataflow (SDF) and Ptides [35] MoCs for co-simulation.

1.2.2 MetroII

MetroII originates from Metropolis, a design framework [4] with the aim of supporting the platform-based design paradigm. Metropolis consists of the metamodel [34] language, a specification language, an infrastructure, and a set of tools for various design activities. The main features include modeling of models of computation, orthogonalization of concerns and the support of both imperative code and declarative statements in the specification. In Metropolis, systems can be modeled unambiguously on different abstraction levels, design problems can be formulated mathematically, and tools can be incorporated to automatically solve some of the problems.

MetroII [10] is the successor of the Metropolis design framework. It implements the platform-based design methodology based on a SystemC simulation engine. MetroII allows designers to import models developed using external, domain-specific tools. For example, a SystemC architectural model can be imported with only minor changes to the original model interface. Instrumental to such integration is MetroII's rigorous and general mapping semantics, which we use to bridge the functional and architectural views of a system.

Specifically, a model in MetroII consists of a set of *MetroII components*. MetroII components are extended SystemC modules that execute concurrently and share data with each other by sending messages via ports. The activations of MetroII components are triggered by MetroII events. Each MetroII event is an extended SystemC event that is managed by the MetroII simulation core. Figure 1.2 shows a simple producer-consumer model in MetroII. A producer component is connected with a consumer component via a FIFO component. The producer and consumer are implemented in two tasks and are mapped to a multi-task single processor architectural model.

Since MetroII is an extension of SystemC, which is a popular system level modeling language for hardware, it allows users to import and design sophisticated architectural models. In addition to that, it also allows users to associate a functional model to an architectural model via mapping. However, MetroII lacks the implementation of the most commonly used MoCs, and it has very limited support for continuous time models, which is important for modeling the physical processes in CPS. In Metronomy, we extend the simulation core of MetroII to support co-simulation with functional models developed in Ptolemy and thus provide an environment that supports commonly used model of computations, continuous time modeling for physical processes, and mapping to high-fidelity sophisticated architectural models.

1.2.3 Other Frameworks

The separation between functional behavior and execution platform is adopted by a number of design frameworks. Model Driven Architecture (MDA) developed by OMG is an architectural framework for software development based on UML. The focus of MDA is to separate the system behavior from the usage of platform. The design starts from a computation independent model (CIM), which captures detailed requirements with no functionality. A

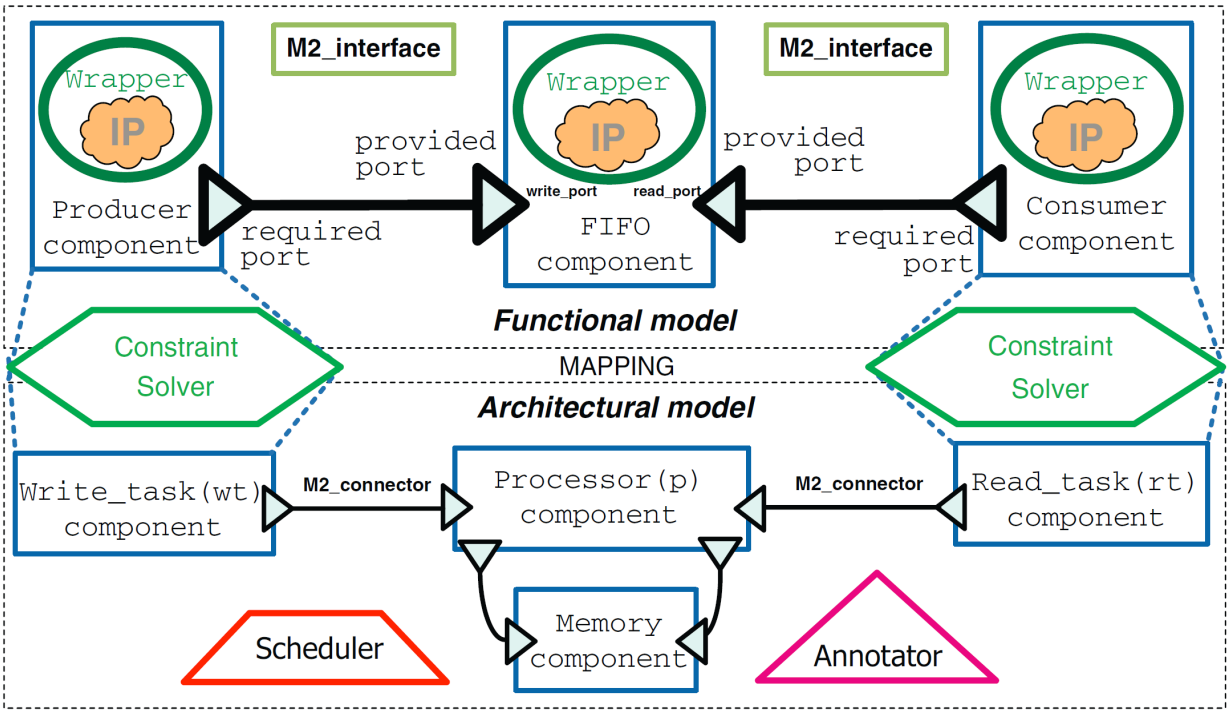

Figure 1.2: A simple MetroII model from [10] consists of a producer and a consumer. They are connected by a FIFO and mapped to two tasks running on a processor.

CIM is then refined into a platform independent model (PIM), which specifies the functionality of the system without dependencies on any particular platform. After that, a PIM is transformed into a platform specific model (PSM) through a mapping that consists of model transformations. And annotations and attributes are used to enrich the PSM model with non-functional properties [8]. The main difference between MDA and Metronomy relates to the focus on architecture exploration. MDA is mainly for software architecture while this dissertation is more focused on hardware architecture modeling. Furthermore, Metronomy employs a mapping that is more generic, and intended to provide performance metrics rather than a detailed implementation. The notion of mapping in this dissertation makes it easier to adapt to different platforms which facilitates the exploration of a large design space.

Model Integrated Computing (MIC) [19] uses domain-specific modeling languages to express the functionality, the architecture and their relation (the mapping). The models are then used to synthesize and integrate the system. The MIC methodology is supported by a set of tools that can create and manage the domain-specific modeling languages. Generic Modeling Environment (GME) [22] has been designed to facilitate the construction and the manipulation of a domain-specific modeling language, by providing a way to specify an abstract as well as a concrete syntax (textual or graphical), including well-formedness constraints and static semantics. MILAN [3, 21] is a verification tool which supports simulator integration using model interpreters, and integrates the design space exploration tool

DESERT [25]. DESERT allows the designer to express platform flexibility by specifying structural constraints in OCL and prune the design space based on these constraints. Unlike GME, this work is not concerned with the design of the modeling language. Instead, we adapt components from different MoCs to our two-phase execution semantics. In addition to that, we are mainly focused on the timing verification in the combined semantics of different models, instead of the relationships between their syntactic elements. Our exploration paradigm also differs substantially from DESERTs. We employ contract theory and timing constraints, instead of structural constraints, and thus are able to relate functionality and architecture without resorting to low level simulators.

SystemC-H [28] is a heterogeneous extension to SystemC [14]. SystemC-H extends the discrete event simulation kernel of SystemC to provide additional MoCs such as dataflow and hierarchical FSMs, using similar techniques as Ptolemy II. The authors demonstrate an increase in simulation efficiency over SystemC with MoC-specific analysis such as static scheduling for dataflow. However, SystemC-H lacks the support from the framework to separate the behaviors and the implementation platform.

ForSyDe [31] initially specifies the system as a deterministic network of fully synchronous processes that communicate over sequences of events. This specification, which lacks detailed timing, is then refined into an implementation by applying a series of network transformations, that may or may not preserve the semantics. [17] extends the basic ForSyDe model to more MoCs and the assumption of a fully synchronous system is dropped in favor of an untimed model similar to Kahn process networks [18]. The transformation-based refinement in ForSyDe has clear advantages in terms of the ability to prove correctness and maintain consistency with the original specification. However, the distinction between functionality and architecture is lost, and a change of mapping may require substantial restructuring of the system. Our approach to mapping, instead, makes this task simpler, since only the mapping function must be changed.

Rosetta [1] describes a MoC declaratively as a set of assertions in a higher order logic. Different MoCs can be obtained by extending a definition in a way similar to the subclassing relation of a type system. Unlike Rosetta, the relationship between the function and the architecture in Metronomy is not described explicitly as a function, but rather as a mapping and annotation process at the event level.

The Behavior-Interaction-Priority (BIP) framework [5] is focused on separating computation and coordination. In BIP, a system specification is divided into three layers. The first layer describes a set of independent components. The second layer controls their activation and interaction via connectors. The top layer overlays a set of priorities to govern component interaction which reduce non-determinism. One of the strengths of the BIP framework is the possibility of checking certain properties, such as deadlock-freedom, compositionally. However, this may require a complex coordination scheme between a large set of connectors. Unlike BIP, we utilize a centralized scheduler to accomplish similar objectives. We also support an imperative or a declarative description for the scheduler while BIP only supports declarative specification of connectors.

Other frameworks include MAPS [9], featured with a variety of mapping heuristics,

Daedalus [26], featured with multi-level design space exploration, Spade [23], featured with a Y-chart based approach, and Sesame [29], featured with trace-based design space exploration. However, most of these frameworks only support Kahn Process Networks (KPN), dataflow or similar MoCs. Furthermore, all previous works are based on the assumption that functional behaviors can be pre-determined, and captured by the functional model, while the implementation platform only affects the system performance. This assumption does not necessarily hold in CPS design, where the function is tightly intertwined with the physical plant (or environment). Different behavioral timings may trigger different reactions from the environment, which result in different further behaviors of the system.

1.3 Contributions

In this dissertation, we present Metronomy, a modeling and co-simulation framework that bridges the functional and the architectural aspects of the design. In Metronomy, the functional model is captured in the Ptolemy modeling environment [30], while the architectural model is described in the MetroII design environment [10]. Ptolemy provides a rich set of commonly used MoCs (e.g. dataflow, state machines, discrete event, discrete time) to effectively model and simulate the system functionality and its interaction with the physical plant. It also allows physical plant to be modeled in continuous time domain. MetroII provides a framework for modeling high-fidelity architecture as well as interfacing functional and architectural models. It allows the architecture to be modeled in SystemC [14]. It also allows third-party architectural models to be imported with minor changes.

Metronomy is a natural framework for multi-domain system engineering and integration. Control engineers can leverage the plethora of MoCs made available by Ptolemy to capture the functionality of their controller; software/hardware engineers can benefit from the flexibility of MetroII and SystemC to design the architectural platform; system engineers can effectively combine the two aspects in a co-simulation environment to explore the whole design space and verify correctness and performance of their design.

To support multi-domain system engineering and integration, we exploit contract-based design theory [33] to facilitate timing verification and design space exploration using co-simulation. A timing contract can be seen as a set of timing assumptions and guarantees that are agreed upon by the control engineers, who develop the functional model, and the software/hardware engineers, who design the architectural platform for implementation. We implement timing checkers in Metronomy to monitor whether both the functional and architectural timing assumptions and guarantees are satisfied during co-simulation.

A preliminary attempt at integrating Ptolemy and MetroII was presented in [20]. However, the integrated design framework in [20] is only able to support a simple Model of Computation (MoC), which schedules actors periodically, and lacks the capability of handling heterogeneous MoCs (e.g. continuous time, discrete time). With respect to [20], this dissertation offers a formalization of the interactions between functional and architectural models in terms of contracts for timing verification and system integration. The use of con-

tracts to analyze the complex coupling of timing and behaviors has been first advocated in [33]. However, while a few rigorous contract theories have been developed over the years (e.g. see [6, 2]), the concrete application of contracts for timing verification in CPS has not been throughly explored. In [11] different types of timing contracts, such as the Logical Execution Time (LET) [16, 13] and Bounded Execution Time (BET) contracts, denoted as "design contracts", are informally presented as a mean to facilitate the independent refinement of functionality and architecture. In this dissertation, based on the theoretical foundations in [6], we propose a formalization of the timing constraints of a design in terms of assume-guarantee contracts, expressed as assertions on system traces, i.e. sequences of events. Such a formalization is general enough to encompass different kinds of design contracts, and suitable for building monitors to validate them via co-simulation.

The contribution of this dissertation is threefold: (a) we formalize the interactions between the functional model and the architectural model via the concept of timing contract; (b) we propose a methodology for timing contract verification and design space exploration through co-simulation; (c) we implement a function-architecture co-simulation framework that supports the methodology, based on the Ptolemy and MetroII design environments.

The rest of this dissertation is organized as follows. Chapter 2 formalizes the concept of timing contract between functional and architectural models and presents the methodology for timing verification and design space exploration using Metronomy. Chapter 3 describes several key aspects of the implementation of Metronomy. Chapter 4 shows its application to the design of an aircraft electrical power system and a paper-feed subsystem of a high-speed printing press. Finally, Chapter 5 presents the conclusions.

Chapter 2

Timing Verification of Cyber-physical Systems

A modern cyber-physical system typically consists of a number of complex subsystems. For example, a vehicle includes a powertrain system, starting and charging system, steering system, suspension system, braking system, etc. Many of these systems are intrinsically heterogeneous. To design such systems, the design team typically consists of engineering members from multiple domains. The control engineers focus on the interactions between the physical processes and computations and the software/hardware engineers focus on the correct implementations of the control algorithms.

Although the design concerns of control engineers and software/hardware engineers should be completely separated, their designs still have impacts on each other in many aspects. Among these aspects, timing is one of the most important aspects. A slight change in timing of the implementation platform may affect the control of physical processes. And the change in the physical processes may in turn affect the behaviors of the implementation platform. The complex interplay across domains not only challenges the engineers but also the design tools that engineers use to analyze and verify the timing of the system. On the one hand, the design tools used by control engineers and software/hardware engineers are completely different. On the other hand, the impacts of timing that couple the function and the implementation platform should be captured, especially in early design stage since these impacts greatly affect the design choices. A strict and general design framework that bridges function and the implementation platform is still missing in the current design process.

In this chapter, we will present a general framework for timing verification that bridges the design tools used by control engineers and software/hardware engineers. Specifically, the control function is modeled in Ptolemy and the implementation platform is modeled in MetroII. We exploit the contract theory to bridge the two aspects and allows engineers to analyze the timing properties of the system. In Section 2.1 and Section 2.2 of this chapter, we briefly discuss how the control function and the implementation platform are modeled. In Section 2.3, we focus on the interactions between the two models and present the timing verification framework. In Section 2.4, we present the methodology for timing verification

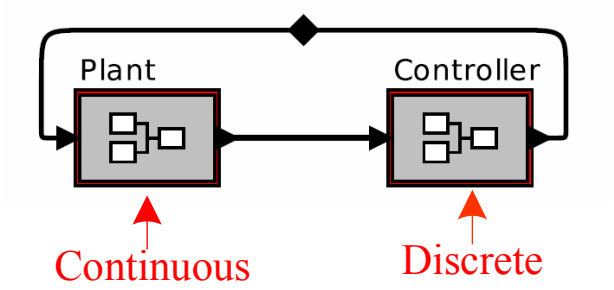

Figure 2.1: The typical model of a cyber-physical system.

and design space exploration using Metronomy.

2.1 Modeling Functional Model in Ptolemy

2.1.1 Representation of Functional Model

For a CPS control design, the functional model is used to describe the control algorithm and its interaction with the physical plant. Due to the intrinsic concurrency of the CPS control model, the actor-oriented modeling is a popular choice in both academia and industry. In an actor-oriented model, each component of the system is implemented as an actor. The actors execute concurrently and exchange data by sending messages via ports.

Figure 2.1 shows the top level of a typical CPS model in Ptolemy. It includes a model of physical plant and a model of controller. The model of physical plant describes the dynamics of physical processes in terms of differential equations and is typically represented in a continuous domain where the time is a continuum. The model of controller describes the control algorithm and is typically represented in a discrete domain where the behaviors of the control algorithm can be captured by timed events.

Figure 2.2 shows an example of such a CPS model in Ptolemy with more details. The model is a simplified controller in a paper feed subsystem of a printing press (see also Section 4.2 for further details). The controller regulates the surface velocity of a roller by adjusting the drive voltage of a motor. The controller has four inputs: tv, ev, mv, ct, and one output cv. Input tv is the profiled target velocity, ev is a real-time adjustment on the profiled target velocity based on the state of the other rollers, mv is the measured velocity of the roller, ct is a signal that turns off the motor, and cv provides the drive voltage to the motor. The controller tries to minimize the error between the measured velocity mv and the target velocity $tv + ev$. When a sporadic signal ct occurs, the controller outputs 0. p_1, p_2, p_3, p_4 are the execution paths from ev, tv, mv, ct to cv, respectively.

Following the semantics of Ptolemy, each component of the functional model in Figure 2.2 is implemented as an actor. A path in the model consists of a cascade of a sensor actor, a series of interconnected execution actors and an actuator actor.

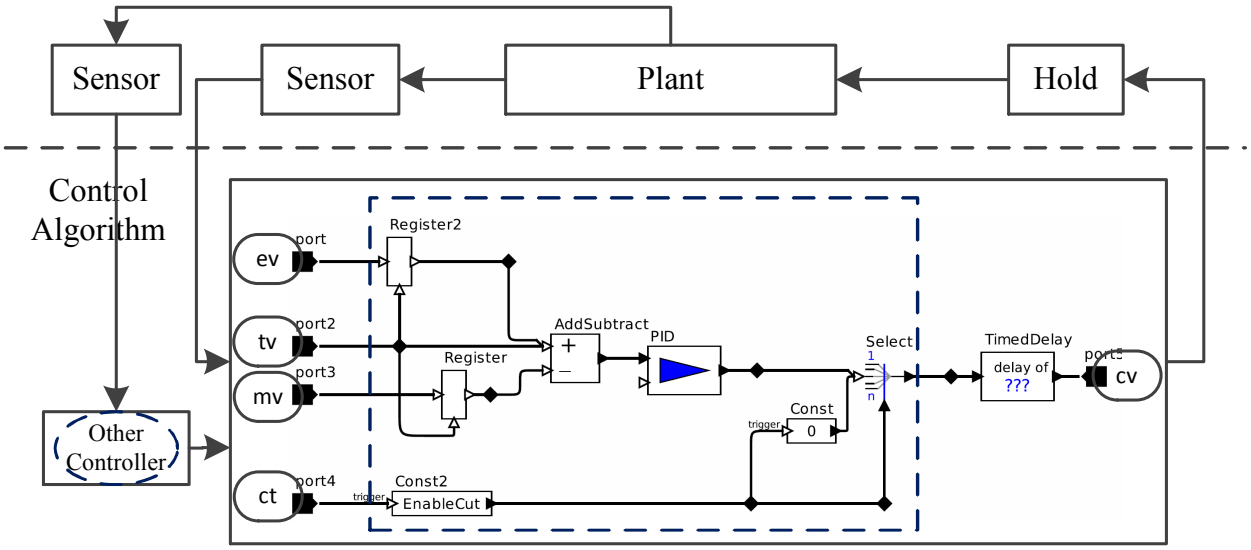

Figure 2.2: A cyber-physical model in function design environment includes the model of a physical plant, the model of sensors, and the model of a control algorithm.

2.1.2 Simulation of Functional Model

The execution of the functional model is based on its Model of Computation (MoC), which is implemented as a director in Ptolemy. For a CPS, the most commonly used directors for modeling the controller are based on discrete events. Each event has a time tag when it is generated. And all the events are ordered and processed chronologically. The director processes the oldest event first and the model time advances to the time tag of the event processed. The processing of an event could trigger an actor to fire. And the firing of an actor may generate more events. From the perspective of an actor, the execution of a model is a process that consists of a series of firings. In each firing, the actor reacts to the events that occur between the previous firing and the current firing, and generate more events. In particular, each message sent between actors is an event which could trigger the firings of actors who receive the message. In Ptolemy, the message is also referred to as *token*. The firing is hierarchical if an actor is a composite actor. The firing of the composite actor will trigger the execution of the enclosed director that fires the enclosed actors according to the MoC defined by the director.

For example, in Figure 2.2, a token received at input port *ev* will trigger the firing of *Register*2. The firing of *Register*2 will generate a token that triggers *AddSubtract*. The firing of *AddSubtract* will generate a token that triggers PID (Proportional-Integral-Derivative). And the firing of PID will generate a token that triggers *Select*. The firing of *Select* will generate a token that triggers $TimedDelay$. The firing of $TimedDelay$ will generate a token that will be sent to output port *cv*. Note the time maintained by the director is advanced only when $TimedDelay$ is fired. In other words, during the series of firings above, only the token generated by $TimedDelay$ has a time tag that is greater than

the triggering token at input ev.

2.1.3 Notions of Time

In this work, we capture three different notions of time in a CPS model: *physical time*, *logical time*, and *platform time*. Physical time represents the idealized time throughout the system. It is the time notion in the model of physical plant to describe the dynamics of physical processes. Physical time is also used in the model of sensors and actuators to capture the interaction between the controller and the plant. Logical time is a time notion first introduced in distributed system to achieve deterministic system behaviors. It gives the order of two events without binding with specific points of physical time. In other words, logical time defines the relationship of "happened before" between events. Platform time is based on the clock on the implementation platform that measures the physical time and thus is subject to a clock drift.

In the environment where the functional model is developed, designers have the notions of physical and logical time. Specifically, in Ptolemy, the physical time and logical time are realized by $DEDirector$ and $PtidesDirector$ respectively. A typical pattern is two-layer structure. On the top level of a Ptolemy model, we use a $DEDirector$ that maintains physical time to capture the interactions between controllers and physical plants. And inside each composite actors that represents a controller, we use a $PtidesDirector$ that maintains logical time to capture the event order in the control algorithm. The conversion between physical time and logical time is handled by input and output ports of the composite actor that represents the controller. For example, the top level in Figure 2.1 uses a $DEDirector$ and the controller modeled in Figure 2.2 uses a $PtidesDirector$.

Modeling the controller in logical time gives the flexibility for software/hardware engineer to realize the design. If physical time is used in modeling the controller, the design of controller is essentially tied to detailed timed behaviors. The software/hardware engineers then have no choice but to precisely implement the detailed timed behaviors without any flexibility. We argue that this would result in inefficient implementation. The physical time should only be used in modeling the physical plant and sensors/actuators.

2.2 Modeling Architectural Model in MetroII

2.2.1 Representation of Architectural Model

In Platform-based Design (PBD), the *functional model* defines what the design does in terms of a set of services and the *architectural model* describes how these services are implemented by a collection of architectural primitives. In this work, the implementation platform is the architecture that realizes the controllers in a cyber-physical system. It describes how the services used to describe control algorithms are implemented in terms of a set of architectural primitives. More specifically, for a CPS model in Ptolemy, the control algorithm is

implemented in terms of actors from the Ptolemy library, where each actor can be seen as a service. The model of implementation platform describes how these services are implemented on a particular platform in terms of software/hardware. For example, for the control algorithm shown in Figure 2.2, the model of implementation platform should at least describe the implementations of actors *Register*, *Const*, *AddSubtract*, *PID*, *Select*, etc in terms of software/hardware.

Furthermore, the way to implement a control algorithm is not unique. The software/hardware engineers should have the freedom to choose the appropriate granularity. Specifically, for a CPS model in Ptolemy, the service implemented on the particular platform could realize an atomic actor or a composite actor. For example, for the controller in Figure 2.2, instead of implementing actors *Register*, *Const*, *AddSubtract*, *PID*, *Select*, the software/hardware engineers may choose to implement the control algorithm as a single service on the implementation platform.

The main use of the architectural model of this work is to give the detailed timing information of the behaviors of the functional model on a particular implementation platform. It can be as simple as a lookup table or as complex as a sophisticated simulator. For example, if the services of *Register*, *Const*, *AddSubtract*, *PID*, and *Select* are implemented on an architecture, the architectural model for the functional model in Figure 2.2 could be seen as a function as follows:

$$f(Serv, Arch) \tag{2.1}$$

where *Arch* is the architecture and *Serv* is the service that executes on the architecture. $f(Register, Arch)$, $f(Const, Arch)$, $f(AddSubtract, Arch)$, $f(PID, Arch)$, and $f(Select, Arch)$ represent the execution time of the services that implement the corresponding actors. When the functional model co-simulated with the architectural model, timed behaviors can be obtained.

However, on most realistic architectures, the execution time is not constant. It is determined by the state of the architecture that keeps changing. The more details we include in the architectural model, the more accurate the architectural model is. In our framework, the execution time of a service could be determined by the following factors:

- The execution time is determined by the services on the lower level. For example, the execution time of *AddSubtract* is determined by the design of the processor, which relies on the circuits that carries on the operation. From the perspective of Platform-based Design, *AddSubtract* is the function, which is implemented by the services on the lower-level architecture. The lower levels we go to, the more accurate our architectural model is.

- The execution time is determined by the shared computation/communication resources and the scheduling mechanism on the implementation platform. With limited resources, many behaviors that appear to be concurrent in functional model are implemented by sequential executions of services in the architectural model. The way how

the shared resources are scheduled greatly affects whether certain concurrent behaviors can be implemented or how fast a control algorithm is.

- The execution time is determined by the input of the service. For example, if a service computes the following function

$$f(n) = \begin{cases} true & \text{if n is a prime number} \\ false & \text{otherwise} \end{cases}$$

the execution of the service greatly depends on the input n.

In Metronomy, the characteristics of the lower level services are captured by a timing contract that will be discussed in Section 2.3. The shared resources and the scheduling mechanism are captured by the architectural model. And input of the service can be also captured by a general contract [33]. In addition to that, a good architectural modeling framework should allow engineers to build a detailed architectural model that is close to the real implementation but still easy to connect to the functional model. MetroII is a suitable candidate because: 1) it is built based on SystemC, an industrial standard for system level modeling of software/hardware. This allows software/hardware engineers to work in the most natural environment; 2) MetroII natively supports structural mapping, which connects to the functional model.

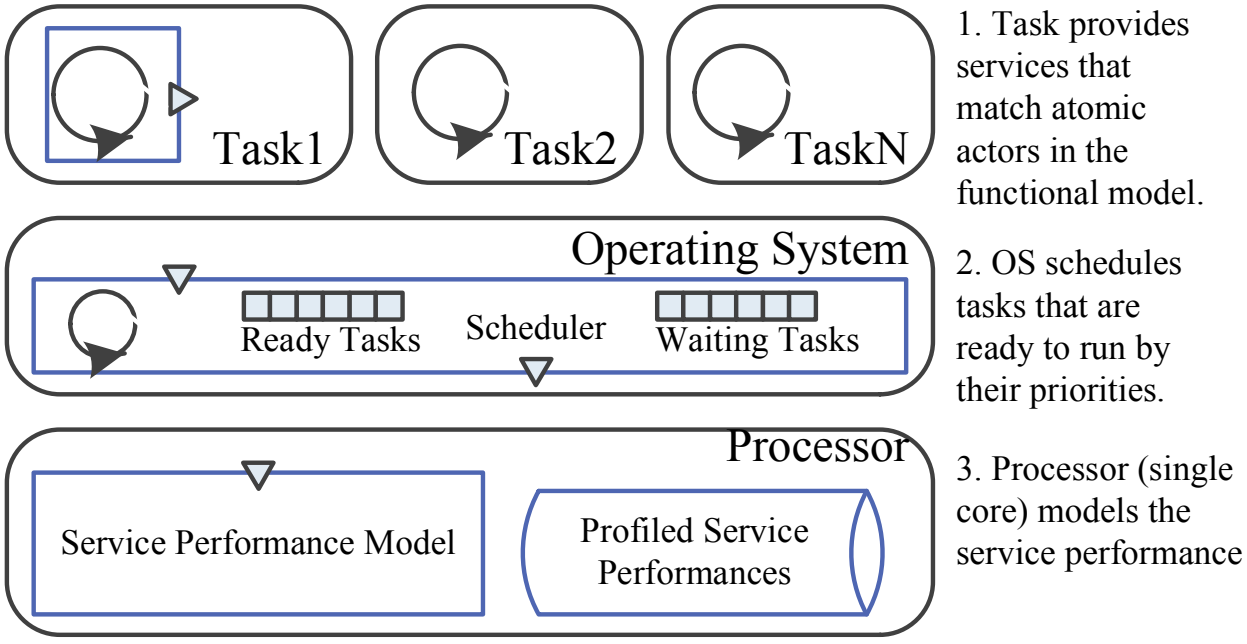

Figure 2.3: The model of implementation platform with a single-core processor and a multi-task OS.

Figure 2.3 shows an example of architectural model in MetroII which has the three layers. On the task layer, each task is a MetroII component that is driven by a SystemC thread.

And the services that implements the atomic actors of the functional model are provided in the task. For example, for the functional model in Figure 2.2, a task can execute one of the firings of *Register*, *Const*, *AddSubtract*, *PID*, and *Select* at a time. A task serves as a mapping target for the composite or atomic actor in the functional model. And each task can be blocked by other tasks that have higher priorities or the Operating System.

The Operating System is also a MetroII component driven by a SystemC thread. In this model, it serves as an explicit, imperative program for managing processor resource, scheduling and assigning tasks to the processor. It maintains a list of tasks that are ready to run and a list of tasks that are waiting. When a waiting task has a service ready to run, the task is inserted into the list of tasks that are ready to run. When a task has no more service ready, it is moved back to the list of waiting tasks. When the processor is available, the OS determines which process will run in the next time slot from the ready list by the task priority. The OS programs the Programmable Interrupt Timer (PIT) to generate an interrupt after N cycles. The task is executed. After N clock cycles, an interrupt occurs and the OS gets a chance to schedule the tasks again. The execution of OS itself also takes certain cycles of the processor.

The processor is also a MetroII component. It is the actual processing element that fulfills the service requests from OS and tasks. In this architectural model, each atomic actor in the Ptolemy is fulfilled by a service. And the OS layer is also fulfilled by a set of services. The processor has a service performance model and a library where precomputed performance metrics are stored for lookup. Similar to Equation 2.1, the performance metrics in the library represent the performance model of the lower level services. When the OS or task indicates which service is requested, the performance model performs a table lookup and triggers a performance metric computation. A performance metric computation at processor level can be seen as the following function:

$$g(Serv, Proc) = \begin{cases} w_{reg} & Serv = Register \\ w_{con} & Serv = Const \\ w_{add} & Serv = AddSubtract \\ w_{pid} & Serv = PID \\ w_{sel} & Serv = Select \end{cases} \tag{2.2}$$

where $Proc$ is the processor and $Serv$ is the service that executes on the processor. The performance metrics (i.e. w_{reg}, w_{con}, w_{add}, w_{pid}, w_{sel}) are computed by one of the following method: 1) a constant number from profiling; 2) a mathematical calculation based on the current state of the processing element; 3) a runtime processing where the pre-compiled code is loaded and executed at runtime which returns the cycle requirements for that code.

To sum up, this example of architectural model gives the performance estimation of the services on task level by a simulation that uses the performance estimation of the services on processor level. The relationship between two levels is summarized in the following formula:

$$f(Serv, Arch) = g(Serv, Proc) + w_{blocking} + w_{overhead} \tag{2.3}$$

Table 2.1: MetroII events that indicate the beginning and ending of services.

Service	Beginning Event	Ending Event
Register	*reg.b*	*reg.e*
Const	*con.b*	*con.e*
AddSubtract	*add.b*	*add.e*
PID	*pid.b*	*pid.e*

where $w_{blocking}$ is the time of the service being blocked by tasks with higher priorities and $w_{overhead}$ includes the overheads of the OS and the context switching. They are obtained from the simulation.

2.2.2 Simulation of Architectural Model

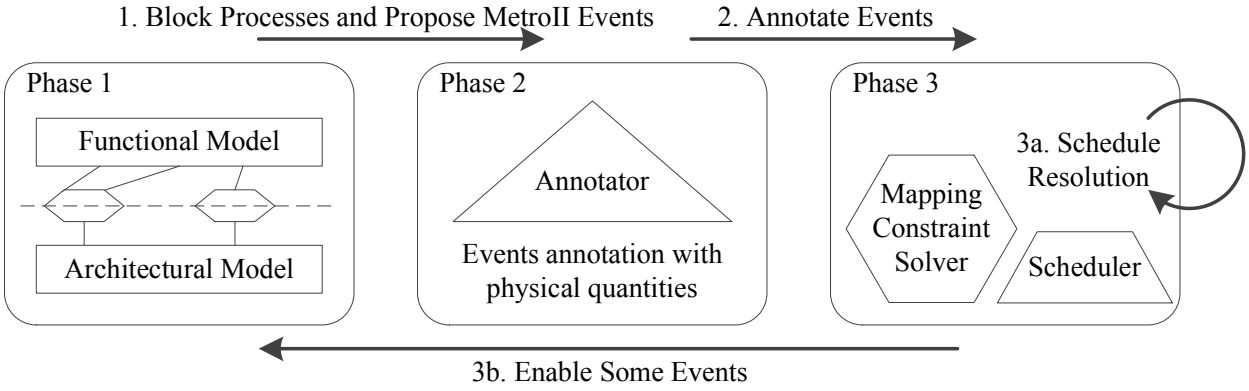

Figure 2.4: Three-phase execution in MetroII.

The execution of MetroII model is based on a simulation core that extends SystemC engine. The MetroII framework wraps interfaces in SystemC and intercepts all the messages about events from SystemC threads before they are sent to SystemC simulation engine. An event has to go through three phases shown in Figure 2.4. In the first phase, the MetroII component executes until it has to trigger an event. This is also referred to as proposing events. The MetroII component is blocked and the message of the proposed event is intercepted by MetroII. In the second phase, MetroII framework annotates time tags to proposed events, which are obtained from the performance model (e.g. Equation 2.3). In the third phase, the proposed events are filtered by a set of constraints and schedulers, a subset of the proposed events are allowed to execute and committed to the SystemC simulation engine.

For example, a task in Figure 2.3 implements services of *Register*, *Const*, *AddSubtract*, *PID*, and *Select*. The architectural model defines two MetroII events for each service as follows: where the beginning event indicates the beginning of execution of the service and

the ending event indicates the end of the execution. In the first phase, when the task starts to execute, it proposes the beginning events of all the services it implements: *reg.b*, *con.b*, *add.b*, and *pid.b*. The task is then blocked. In the second phase, MetroII annotates time tags to these proposed events. In the third phase, after resolving all the constraints about the mapping, one event (for instance, *pid.b*) is allowed to execute. Then it is notified and committed to SystemC engine. In the next iteration, the service of PID will begin executing on the architecture. Note that the task is scheduled by OS. When the task is blocked, no more events in Table 2.1 will be proposed until the current service is executed.

2.2.3 Notions of Time

As discussed in 2.1.3, three different notions of time are captured in a CPS model: *physical time*, *logical time*, and *platform time*. An architectural model typically maintains physical time and platform time, where physical time represents the idealized time throughout the system and platform time is the clock on the implementation platform that measures the physical time. For example, in Figure 2.3, the physical time is the idealized time and platform time is the clock of the processor. Logical time is sometimes used internally in the architectural model and is mapped to physical time.

2.3 Timing Contracts Between Models

As discussed in Section 2.1 and Section 2.2, in our co-simulation framework, a *system model* includes a higher-level *functional model*, a lower-level *architectural model* and a *mapping function*. The functional and architectural models provide two different representations of the system at different levels of abstraction, and can possibly cover different design aspects or *viewpoints*. The mapping function links how the behaviors of the functional model are mapped into behaviors of the architecture during co-simulation.

2.3.1 Definition of Timing Contract

We define a *timing contract* as a tuple $\mathcal{C} = (\mathcal{E}, \mathcal{T}, \mathcal{A}, \mathcal{G})$, where $\mathcal{E}$ is a set of events, $\mathcal{T}$ is a set of time tags, $\mathcal{A}$ is a set of assumptions, and $\mathcal{G}$ is a set of guarantees. We can then denote the interface between the functional model and the architectural model by specifying a functional contract $\mathcal{C}_f = (\mathcal{E}_f, \mathcal{T}, \mathcal{A}_f, \mathcal{G}_f)$, an architectural contract $\mathcal{C}_a = (\mathcal{E}_a, \mathcal{T}, \mathcal{A}_a, \mathcal{G}_a)$, and a mapping function $\mathcal{M}$.

$\mathcal{E}_f$ is a set of events capturing the activity in the functional model, $\mathcal{E}_a$ is a set of events capturing the activity in the architectural model, $\mathcal{T}$ is a set of time tags that define a common notion of time shared by the two models. For each event e ($e \in \mathcal{E}_a$ or $e \in \mathcal{E}_f$), $t_e \in \mathcal{T}$ is its time tag.

An event in the functional model $e \in \mathcal{E}_f$ is represented by a tuple $e = (fun.id, k)$, where *id* is the identifier of the event, which specifies, for instance, the arrival of sensing data, the

beginning or the ending of a computation process, or the application of a certain action; k is an integer index denoting the k-th instance of the event. Similarly, each event $e \in \mathcal{E}_a$ is a tuple $e = (arch.id, k)$. When there is no confusion, we will abbreviate all the events as (id, k).

$\mathcal{A}_f$ and $\mathcal{G}_f$ are, respectively, the set of assumptions made by the functional model, and the set of guarantees provided by the model under the assumptions. Following the formulation in [6], in our framework, both $\mathcal{A}_f$ and $\mathcal{G}_f$ are sets of *behaviors* over $\mathcal{E}_f$. A behavior is defined as a *trace*, i.e. a sequence of events. Sets of traces are captured using assertions including constraints on their event time tags. Similarly, $\mathcal{A}_a$ and $\mathcal{G}_a$ represent, respectively, the sets of assumptions and guarantees related to the architectural model, and can also be expressed as assertions on the time tags of the events in $\mathcal{E}_a$.

More specifically, both assumptions and guarantees can be expressed using first order logic formulas defined as follows. A linear inequality defined on a set of event time tags t_e and other variables is a formula. If α is a formula, then $\neg\alpha$ is a formula. If α_1 and α_2 are formulas, then $\alpha_1 \wedge \alpha_2$ ($\alpha_1 \vee \alpha_2$, $\alpha_1 \rightarrow \alpha_2$) is a formula. If α is a formula and x is a variable, then $\forall x, \alpha$ ($\exists x, \alpha$) is a formula.

Finally, $\mathcal{M}$ maps events in the functional model into events in the architectural model. For a pair of events $e_1 \in \mathcal{E}_f$ and $e_2 \in \mathcal{E}_a$, if $\mathcal{M}(e_1) = e_2$, then $t_{e_1} = t_{e_2}$.

Examples of assertions used to express assumptions and guarantees are provided below:

- End-to-end path latency:
$$\forall k, \quad t_{(p.e,k)} - t_{(p.b,k)} \leq d \tag{2.4}$$
where $(p.b, k)$ is the beginning event corresponding to the k-th computation of path p, $(p.e, k)$ is the ending event corresponding to the k-th computation of path p, and d is the deadline for the path latency.

- Periodic events:
$$\forall k, \quad t_{(id,k+1)} - t_{(id,k)} = T \tag{2.5}$$
where T is the period.

- Sporadic events:
$$\forall k, \quad t_{(id,k+1)} - t_{(id,k)} \geq T_s \tag{2.6}$$
where T_s is the minimum interval of two consecutive events with the same id.

- Partial order of events:
$$t_{e_1} < t_{e_2}, \tag{2.7}$$
which can be used to encode several type of constraints such as the amount of requested computation or data dependencies.

As an example, for a functional model, $\mathcal{A}_f$ could be (2.4), while $\mathcal{G}_f$ could be a conjunction of assertions in (2.5), (2.6) and (2.7). Since we only focus on timing assertions, we assume that the available architecture platform can implement any behavior ($\mathcal{A}_a = True$) at possibly

different costs. On the other hand, $\mathcal{G}_a$ is a set of performance guarantees from the services implemented on the architecture. A typical architecture guarantee on the execution of a service could be

$$t_{(id.e,k)} - t_{(id.b,k)} \leq w \tag{2.8}$$

as in (2.4), where w is now the execution time for service id. In a simple contract, the parameters (e.g. w) in the assertions above could be the worst execution time of the service. In a complex contract, the parameters are not necessarily constants but can be variables that dynamically change during the simulation. Generally, w depends on processor speed, bus speed, cache size and cache strategy etc. And if the execution of the service on the platform is not atomic, it also depends on preemptions of other tasks with higher priorities.

2.3.2 Illustrating Example

To illustrate how timing contracts can be applied to a control system, we consider the system in Figure 2.5 again, which is a simplified controller in a paper feed subsystem of a printing press (see also Section 4.2 for further details). The controller regulates the surface velocity of a roller by adjusting the drive voltage of a motor. The controller has four inputs: tv, ev, mv, ct, and one output cv. Input tv is the profiled target velocity, ev is a real-time adjustment on the profiled target velocity based on the state of the other rollers, mv is the measured velocity of the roller, ct is a signal that turns off the motor, and cv provides the drive voltage to the motor. The controller tries to minimize the error between the measured velocity mv and the target velocity $tv + ev$. When a sporadic signal ct occurs, the controller outputs 0. p_1, p_2, p_3, p_4 are the execution paths from ev, tv, mv, ct to cv, respectively. By following the semantics of Ptolemy, each component of the functional model in Figure 2.5 is implemented as an actor. Strictly speaking, a path in the model consists of a cascade of a sensor actor, a series of interconnected execution actors and an actuator actor. In this example, we assume the delays of the sensor and the actuator are zero for simplicity and thus the end-to-end latency is the sum of delays due to the execution actors along the path.

We denote the beginning event and the ending event of the k-th computation of path p_i as $(p_i.b, k)$ and $(p_i.e, k)$ respectively. If an output is generated in the k-th computation, the output occurs at time $t_{p_i.e,k}$. l represents the latency between the activation of the controller and the output. Different l may deliver different control performances.

Events $(c.b, k)$ and $(c.e, k)$ are the beginning event and the ending event of the k-th firing of the controller, which carries on the computation of its paths. Note that one firing of the controller may carry on the computations of multiple paths. The firing of the controller is triggered by the events at the input ports. Between $(c.b, k)$ and $(c.e, k)$, there are also events indicating the beginnings or the endings of the firings of internal actors. For example, $(c.pid.b, k)$ and $(c.pid.e, k)$ indicate the beginning and the ending of one firing of the internal actor PID (Proportional-Integral-Derivative).

The assumptions of the functional contract $\mathcal{C}_f$ are specified by the conjunction of assertions on the end-to-end latencies of paths. For example,

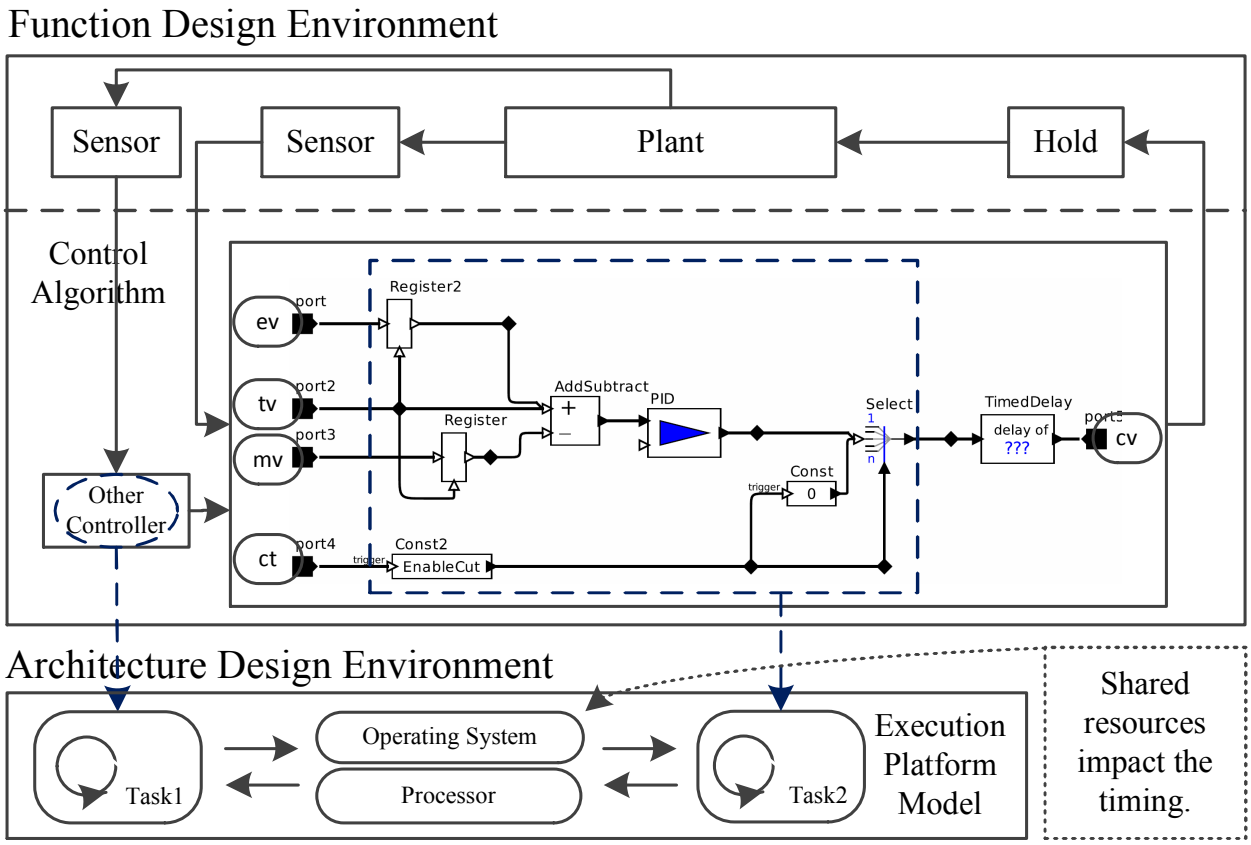

Figure 2.5: A simplified controller in a printing press paper feed system. Controllers in the function design environment are mapped to a single processor multi-task execution platform in the architecture design environment.

- *End-to-end latency* of path p_1:

$$\begin{aligned} &\forall k, && t_{(p_1.e,k)} - t_{(p_1.b,k)} = l \leq d_1 \\ &\forall k_1, k_2, && t_{(p_1.b,k1)} = t_{(c.b,k_2)} \rightarrow t_{(p_1.e,k_1)} = t_{(c.e,k_2)} \end{aligned}$$

  End-to-end latency of path p_2:

$$\begin{aligned} &\forall k, && t_{(p_2.e,k)} - t_{(p_2.b,k)} = l \leq d_2 \\ &\forall k_1, k_2, && t_{(p_2.b,k_1)} = t_{(c.b,k_2)} \rightarrow t_{(p_2.e,k_1)} = t_{(c.e,k_2)} \end{aligned}$$

  End-to-end latency of path p_3:

$$\begin{aligned} &\forall k, && t_{(p_3.e,k)} - t_{(p_3.b,k)} = l \leq d_3 \\ &\forall k_1, k_2, && t_{(p_3.b,k_1)} = t_{(c.b,k_2)} \rightarrow t_{(p_3.e,k_1)} = t_{(c.e,k_2)} \end{aligned}$$

  End-to-end latency of path p_4:

$$\begin{aligned} &\forall k, && t_{(p_4.e,k)} - t_{(p_4.b,k)} = l \leq d_4 \\ &\forall k_1, k_2, && t_{(p_4.b,k_1)} = t_{(c.b,k_2)} \rightarrow t_{(p_4.e,k_1)} = t_{(c.e,k_2)} \end{aligned}$$

which state that the execution time l of the controller that carries the computations of one or multiple paths must complete before the deadlines d_1, d_2, d_3, and d_4. d_1, d_2, d_3, and d_4 represent the requirements on end-to-end latencies of paths p_1, p_2, p_3, and p_4. These requirements are typically associated with the requirements from the physical plant.

The guarantees of the functional contract $\mathcal{C}_f$ are also specified by a conjunction of assertions. Examples of assertions are

- *Controller activation*:

$$\begin{aligned}\forall k,\, t_{(p_1.b,k)} &= t_{(ev,k)}\\ \forall k,\, t_{(p_2.b,k)} &= t_{(tv,k)}\\ \forall k,\, t_{(p_3.b,k)} &= t_{(mv,k)}\\ \forall k,\, t_{(p_4.b,k)} &= t_{(ct,k)}\end{aligned}$$

  guarantee that computation of paths p_1, p_2, p_3 and p_4 are triggered by input signals ev, tv, mv, and ct.

$$\begin{aligned}\forall k_1, \exists k_2, t_{(p_1.b,k_1)} &= t_{(c.b,k_2)}\\ \forall k_1, \exists k_2, t_{(p_2.b,k_1)} &= t_{(c.b,k_2)}\\ \forall k_1, \exists k_2, t_{(p_3.b,k_1)} &= t_{(c.b,k_2)}\\ \forall k_1, \exists k_2, t_{(p_4.b,k_1)} &= t_{(c.b,k_2)}\end{aligned}$$

  guarantee that there is always a firing of the controller that carries on the computation of path p_1, p_2, p_3 and p_4.

- *Periodic event* tv, ev, and mv:

$$\begin{aligned}\forall k, t_{(ev,k+1)} - t_{(ev,k)} &= T_{ev}\\ \forall k, t_{(tv,k+1)} - t_{(tv,k)} &= T_{tv}\\ \forall k, t_{(mv,k+1)} - t_{(mv,k)} &= T_{mv}\end{aligned}$$

  guarantee that ev,tv, and mv are periodic inputs.

- *Sporadic event* ct: $\forall k, t_{(ct,k+1)} - t_{(ct,k)} \geq T_{ct}$ guarantees that the minimal interval of two consecutive ct is T_{ct}.

- *Amount of requested computation*:

$$\begin{aligned}
&\forall k, \forall j_1, j_2, (t_{(c.b,k)} \leq t_{(c.reg.b,j_1)} \wedge t_{(c.reg.e,j_1+r_1-1)} \leq t_{(c.e,k)} \\
&\quad \wedge t_{(c.b,k)} \leq t_{(c.reg2.b,j_2)} \wedge t_{(c.reg2.e,j_2+r_2-1)} \leq t_{(c.e,k)}) \\
&\quad \rightarrow r_1 + r_2 \leq r_{c.reg} \\
&\forall k, \forall j_3, (t_{(c.b,k)} \leq t_{(c.add.b,j_3)} \wedge t_{(c.add.e,j_3+r_3-1)} \leq t_{(c.e,k)}) \\
&\quad \rightarrow r_3 \leq r_{c.add} \\
&\forall k, \forall j_4, j_5, (t_{(c.b,k)} \leq t_{(c.con.b,j_4)} \wedge t_{(c.con.e,j_4+r_4-1)} \leq t_{(c.e,k)} \\
&\quad \wedge t_{(c.b,k)} \leq t_{(c.con2.b,j_5)} \wedge t_{(c.con2.e,j_5+r_5-1)} \leq t_{(c.e,k)}) \\
&\quad \rightarrow r_4 + r_5 \leq r_{c.con} \\
&\forall k, \forall j_6, (t_{(c.b,k)} \leq t_{(c.pid.b,j_6)} \wedge t_{(c.pid.e,j_6+r_6-1)} \leq t_{(c.e,k)}) \\
&\quad \rightarrow r_6 \leq r_{c.pid} \\
&\forall k, \forall j_7, (t_{(c.b,k)} \leq t_{(c.sel.b,j_7)} \wedge t_{(c.sel.e,j_7+r_7-1)} \leq t_{(c.e,k)}) \\
&\quad \rightarrow r_7 \leq r_{c.sel}
\end{aligned}$$

  where $r_{c.reg}$ specifies the bound for the number of firings of the *Register* actors (including *Register* and *Register2* in Figure 2.5). Similarly, $r_{c.add}$, $r_{c.con}$, $r_{c.pid}$ and $r_{c.sel}$ specify the bounds for the number of firings of each type of actor during one firing of the controller.

Finally, the guarantees of the architectural contract $\mathcal{C}_a$ are specified by the conjunction of assertions on the execution time of services. The assertions on the computations of *Register*, *AddSubtract*, *Const*, *PID* and *Select* would be:

$$\begin{aligned}
&t_{(task2.reg.e,k)} - t_{(task2.reg.b,k)} \leq w_{reg} \\
&t_{(task2.add.e,k)} - t_{(task2.add.b,k)} \leq w_{add} \\
&t_{(task2.con.e,k)} - t_{(task2.con.b,k)} \leq w_{con} \\
&t_{(task2.pid.e,k)} - t_{(task2.pid.b,k)} \leq w_{pid} \\
&t_{(task2.sel.e,k)} - t_{(task2.sel.b,k)} \leq w_{sel}
\end{aligned}$$

where w_{reg}, w_{add}, w_{con}, w_{pid} and w_{sel} are variables depending on the processor speed and the state of the cache. If the service is not an atomic operation on the platform, it also depends on possible preemptions of $Task1$ which has a higher priority.

The architecture assumption is *True* in the example because the guarantees do not rely on any assumption on the functional model.

2.4 Timing Verification and Design Exploration Methodology

Metronomy can be used for timing verification as well as design space exploration. We denote the functional component (model) $Func$ as the set of all the possible traces $\{tr_{f1}, tr_{f2}, ...\}$ defining its behavior, where each trace tr_{fi} is an infinite sequence of events $(e_{fi_1}, e_{fi_2}, \ldots)$. Similarly, the architectural component $Arch$ can be seen as a set of traces. Then, the mapping function $\mathcal{M}$ corresponds to a set of rendezvous constraints on events in the two models: $t_{e_f} = t_{e_a}$ if $\mathcal{M}(e_f) = e_a$.

Given the timing contracts $\mathcal{C}_f$ and $\mathcal{C}_a$, and a system level specification in the form of a contract $\mathcal{C}_s$, the *timing verification* problem translates into checking whether $\mathcal{C}_s$ and the composition of $\mathcal{C}_f$ and $\mathcal{C}_a$ are satisfied by the behaviors of the *mapped model* $Func \times Arch|_{\mathcal{M}}$, i.e. the system model obtained by mapping function behaviors into architecture behaviors. Formally,

$$\begin{aligned} Func \times Arch|_{\mathcal{M}} &\models \mathcal{C}_f \otimes \mathcal{C}_a \\ Func \times Arch|_{\mathcal{M}} &\models \mathcal{C}_s \end{aligned} \tag{2.9}$$

where $Func \times Arch|_{\mathcal{M}}$ is a set of traces including both function and architecture events. Each trace is, in general, an infinite sequence of events, obtained by merging a trace in $Func$ and a trace in $Arch$ that satisfy the rendezvous constraints specified by $\mathcal{M}$. A component (or a system) satisfies a contract (denoted by $\models$ in (2.9)) when its event time tags satisfy the guarantees in the context of the assumptions, i.e., for the functional model,

$$Func \cap \mathcal{A}_f \subseteq \mathcal{G}_f, \tag{2.10}$$

where $Func \cap \mathcal{A}_f$ represent the function behaviors that satisfy the assertions of $\mathcal{A}_f$.

As in [6], the assumptions and guarantees of the composite contract $\mathcal{C}_f \otimes \mathcal{C}_a$ can be defined as follows:

$$\begin{aligned} \mathcal{G}_\otimes &= \mathcal{G}_f \cap \mathcal{G}_a \\ \mathcal{A}_\otimes &= \mathcal{A}_f \cap \mathcal{A}_a \cup \neg \mathcal{G}_\otimes, \end{aligned}$$

where $\neg$ denotes the complement of a set, and all the assumptions and guarantees are assumed to be extended to the same set of variables including both function and architecture events, via a reverse projection operation. If (2.9) hold then we can also conclude that $\mathcal{C}_f$ and $\mathcal{C}_a$ are *consistent*, i.e. there exists an implementation that satisfies both contracts. In Metronomy, we check all the assertions of the composite contracts using monitors during co-simulation of the functional and the architectural models.

In addition to timing verification, we can use Metronomy to perform *design space exploration*, by using timing checkers in an optimization loop, where an objective function (or a set of objectives) is optimized. As an example, if $Func$, $Arch$, $\mathcal{C}_f$, or $\mathcal{C}_a$ are expressed in

parametric form, the design space exploration problem can be formulated as follows:

$$\min_{x_f, x_a, x_c} \; J(Func(x_f), Arch(x_a))$$

$$\text{s.t.} \begin{cases} Func(x_f) \times Arch(x_a)|_{\mathcal{M}} \models \mathcal{C}_f(x_f, x_c) \otimes \mathcal{C}_a(x_a, x_c) \\ Func(x_f) \times Arch(x_a)|_{\mathcal{M}} \models \mathcal{C}_s(x_f, x_c) \\ x_f \in X_f, x_a \in X_a, x_c \in X_c \end{cases}$$

where x_f and x_a are sets of parameters that encode design choices in the functional and the architectural model, respectively; x_c are parameters of the contracts (e.g. see d, $r_{c.reg}$, $r_{c.pid}$ in the illustrating example) which allow relaxing or tightening design requirements; J is cost function. The models obtained from the optimization process, $Func^*$ and $Arch^*$, can then be provided as specifications to be independently implemented (refined) by the control and the embedded system engineers.

Chapter 3

Metronomy Design Framework

Metronomy is a co-simulation framework that integrates Ptolemy and MetroII. It benefits from the heterogeneous modeling and simulation from Ptolemy as well as the multi-level architecture mapping and evaluation from MetroII. In this chapter, we will discuss the implementation of Metronomy.

As shown in Figure 3.1, a Metronomy model includes three components, i.e. the functional model, the architectural model and the mapping. At the top level, the functional model and the architectural model are composite actors governed by the co-simulation director *CoSimDirector*, which extends the execution semantics of MetroII. The mapping is a set of mapping constraints specified in a file that configures the *CoSimDirector*.

The functional model is an actor-based hierarchical model inside a composite actor. For CPS, a discrete event director *CoSimDEDirector* is used to govern the execution of the functional model. Under *CoSimDEDirector*, the physical plant is modeled using directors from continuous-time domain (e.g.*Continuous Director*), while the controller is modeled discrete MoCs (e.g.*CoSimPtidesDirector*), which is adapted to the co-simulation execution semantics.

The architecture is a MetroII model, which is based on a SystemC simulation engine. The model is compiled with SystemC and MetroII libraries into an executable, which then runs in a separate process during co-simulation. The running model is wrapped by a composite actor using inter-process communication. The architectural model also has a configuration file that contains the its parameters.

The mapping function $\mathcal{M}$ is implemented as a set of mapping constraints used by the co-simulation director. A mapping constraint is a rendezvous constraint on a pair of events, where each event is specified by its name. Figure 3.1 also shows examples of mapping constraints, in which the beginning and the ending of firings of PID and *AddSubtract* actors in the functional model are mapped to the beginning and the ending of the PID and *AddSubtract* services of $Task1$ in the architecture.

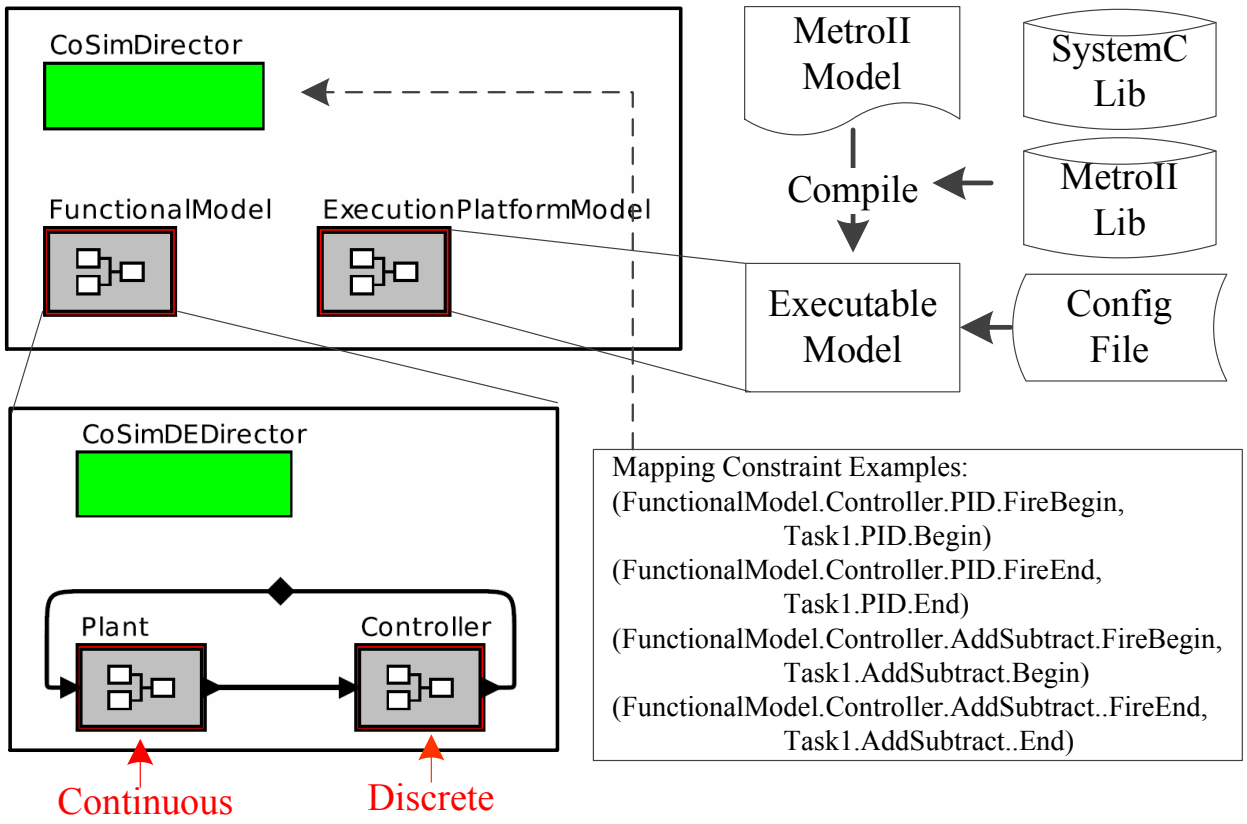

Figure 3.1: A CPS model in Metronomy.

3.1 Co-simulation Director

Each actor under the co-simulation director is either a functional model or an architectural model, which consists of a set of concurrent processes from the perspective of co-simulation director. The simulation progress of each model is controlled by the co-simulation director via events.

An event $e = (id, k)$ is associated to a tuple (id, t, s, V), where id is the event identifier, $t \in \mathcal{T}$ is its time tag (particularly, $t = null$ for un-timed events), $s \in \{proposed, waiting, notified\}$ is the state of the event, and V is a set of additional values that can be used for passing messages between actors (models). Each event marks the beginning or the ending of an activity (e.g. the arrival of sensing data, the beginning or the ending of a computation process, or the application of a certain action). When the event is passed from an actor to the co-simulation director, the state is set to *proposed*, representing an activity "may happen" in the functional or architectural model. The co-simulation director processes the *proposed* events, updates the state to either $notified$ or *waiting*, and passes the event back to the actor. The states $notified$ and *waiting* represent whether the activity associated with the event can proceed or not. Multiple events can be proposed by

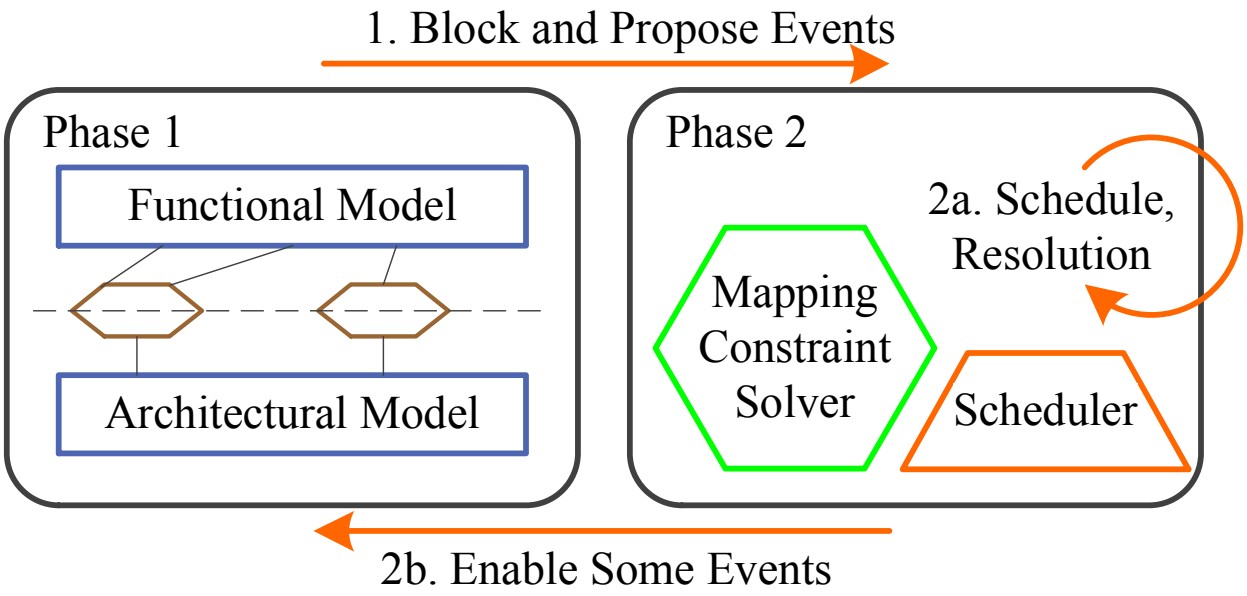

Figure 3.2: The two-phase execution semantics.

one actor to represent the concurrency.

For example, Task2 in Figure 3.3 in architectural model may propose events $task2.reg.b$, $task2.add.b$, $task2.con.b$, $task2.pid.b$, and $task2.sel.b$ because any of them may happen. Then, if the proposed event $task2.pid.b$ is $notified$, the associated activity, service PID, will begin executing if it is allowed by the OS. And other proposed events are set to *waiting* by the co-simulation director, the associated activity will have to wait.

Figure 3.2 shows the execution semantics of the co-simulation director, which is articulated into two phases:

- *Phase* 1: *Base Model Execution*. Each top-level actor (functional or architectural model) executes until it blocks after proposing events. After all the top-level actors block the simulation transitions to phase 2.

- *Phase* 2: *Scheduling or Constraint Solving*. The states of the proposed events are updated based on the resolution of the mapping constraints. A subset of the proposed events are enabled and their states are updated to $notified$, which simultaneously allows their associated composite actors to resume. The rest of the events remain suspended, i.e. their states are updated to *waiting*.

Inside a composite actor, the internal actors are organized hierarchically and each actor is seen as a separate process which is scheduled by the governing MoC director as well as the co-simulation director. In phase 1, an actor has a chance to propose events only when the actor is scheduled by the governing MoC director; an actor can proceed to the phase 1 of the next round only when the proposed events are $notified$ in phase 2.

3.2 Co-simulation Actor

To adapt the function-architecture co-simulation, we have to also change the implementation of actors. Because as explained in 3.1, the firings of the actors in the functional model are governed by not only the director on the same level but also the co-simulation director. In order to achieve this without changing the internal code of each actor, we create a wrapper for each actor. The wrapper exchanges messages with the director that governs the actor in terms of events. Each message contains one or multiple events with states ($\{proposed, waiting, notified\}$). For a particular event, its state can be accessed by the director that directly governs the associated actor as well as $CoSimDirector$ on the top level. If the event is allowed to take place by the implemented MoC of the governing director, it is passed to co-simulation director. Based on the event state given by the co-simulation director, the wrapper of each actor controls the firing of the actor.

We implement two types of wrappers. One is for atomic actor. The other is for composite actor. For an atomic actor, the wrapper allows the actor to be mapped to a service on the architecture. For a composite actor, the wrapper not only allows the composite actor to be mapped but also allows the actors enclosed in the composite actor to be mapped if they are wrapped as well.

3.2.1 Atomic Co-simulation Actor and Deferred Firing

A wrapped atomic actor in the functional model that is allowed to be mapped to the architectural model is an atomic co-simulation actor. As explained in 3.1, the firing of an atomic co-simulation actor is initiated by the governing director on the same level but needs to be approved by the co-simulation director. So the firing in the co-simulation is often deferred in terms of physical time, compared to the firing in a purely functional model without interferences of the architectural model. Because the mapped service on the architecture might be blocked due to preemption or shortage of computing resources. Even with enough priority and computing resources, the firing of an atomic co-simulation actor can still be deferred because it takes time to execute a service on the architecture. The firing in the functional model can only take place when the execution on the architecture completes.

The deferred firing of atomic actor exposes a possible semantic difference between the functional model and the architectural model. If the atomic actor is scheduled by a director with physical time, it has to fire at exactly the point of physical time it is scheduled to. Any deferred firing will violate the semantics of the functional model.

However, this issue can be resolved by scheduling the actors with logical time. Even a firing is deferred in terms of physical time, the deferred firing can still appear to take place at the same logic time it is scheduled to. In Metronomy, we always map the atomic co-simulation actors scheduled by the director with logic time, i.e. PtidesDirector. If we map an atomic co-simulation actor scheduled by a director with physical time, we will instantly get an error unless the architecture can execute the mapped service at the scheduled physical time with no time elapsed.

Scheduling actors by logic time implicitly introduces timing assertions in functional assumption $\mathcal{A}_f$ as the logic time has to be mapped to physical time ultimately. We will discuss in more details in 3.2.2.

3.2.2 Composite Co-simulation Actor and Decomposed Firing

A wrapped composite actor that allows its enclosed actors to be mapped to the architectural model is a composite co-simulation actor. Ptolemy has two types of composite actors. If the composite actor does not enclose any director, the enclosed actors are governed by the governing director of the composite actor. If the composite actor encloses a director, the enclosed actors are governed by the enclosed director. Since the composite actor without enclosed director is only a structure that does not carry any semantics, we only discuss the composite actor that encloses a director.

Without any architectural model, the firing of the composite actor is a sequence of firings of the enclosed actors that follows the semantics of the enclosed director. Suppose all the enclosed actors are atomic co-simulation actors. With an architectural model, the enclosed actors are mapped to the services on the architecture and thus the firings of the enclosed actors are deferred by the execution of the services. Different atomic actors may be deferred by the different amount of time. Consequently, one firing of the composite actor in purely functional model is decomposed into multiple firings of the composite co-simulation actor in the mapped model that follow not only the semantics of the enclosed director but also the execution of services on the architecture.

We use the same example in Figure 3.3 to illustrate the firing of the composite co-simulation actor. The controller is a composite co-simulation actor inside which the enclosed actors are mapped to the services of $Task2$ on the architecture. Suppose one firing of the composite actor is triggered by signals ev, tv and mv simultaneously at time t_0. Without mapping to any architectural model, the firing of the composite actor in the purely functional model is as follows:

- At physical time t_0 (logical time t_0), the firing of the composite actor consists of a sequence of firings of $Register2$, $Register$, $AddSubtract$, PID, $Select$, $TimeDelay$.
- At physical time $t_0 + d$ (logical time $t_0 + d$), the firing of the composite actor gives the output signal.

where d is the time interval specified in $TimeDelay$.

With the architecture in Figure 2.5, the same firing in the purely functional model is decomposed into five firings of the composite co-simulation actor in the mapped model:

- At physical time t_1 (logical time t_0), the firing of the composite co-simulation actor consists of the firing of $Register2$;
- At physical time t_2 (logical time t_0), the firing of the composite co-simulation actor consists of the firing of $Register$;

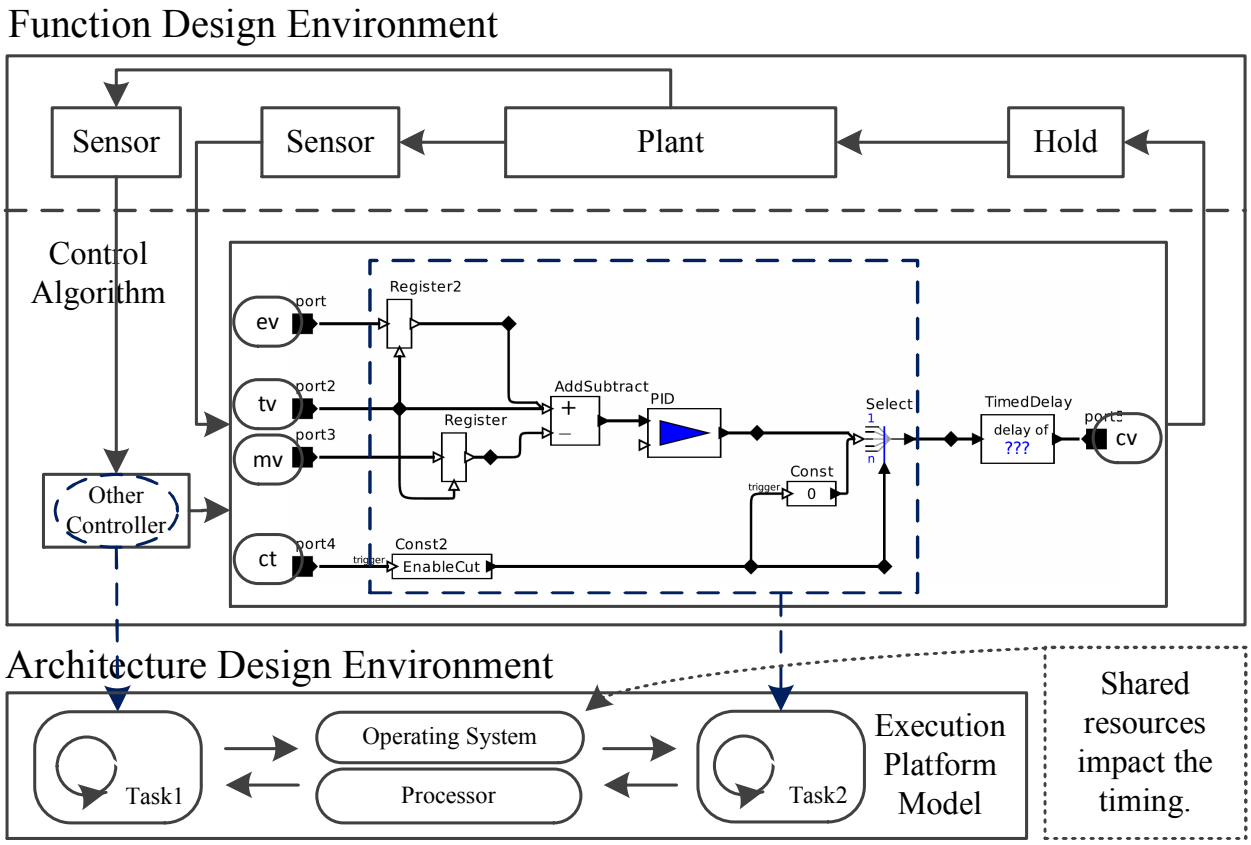

Figure 3.3: A simplified controller in a printing press paper feed system. Controllers in the function design environment are mapped to a single processor multi-task execution platform in the architecture design environment.

- At physical time t_3 (logical time t_0), the firing of the composite co-simulation actor consists of the firing of *AddSubtract*;
- At physical time t_4 (logical time t_0), the firing of the composite co-simulation actor consists of the firing of PID;
- At physical time t_5 (logical time t_0), the firing of the composite co-simulation actor consists of the firing of *Select* and $TimeDelay$.
- At physical time $t_0 + d$ (logical time $t_0 + d$), the firing of the composite actor gives the output signal.

where $t_1 - t_0$ is the sum of the execution time of *Register* service in $Task2$, the possible blocking time of $Task2$ and overheads of the OS. Similarly, $t_2 - t_1$ ($t_3 - t_2$, $t_4 - t_3$, and $t_5 - t_4$) is the sum of execution time of corresponding service, the possible blocking time of $Task2$ and overheads of the OS.

As shown in the above example, the logical time is employed by the enclosed director in the composite co-simulation actor to avoid the semantic difference between the purely functional model and the function-architecture mapped model. No matter how many times the composite co-simulation actor fires, the logical clock only advances when the actor of $TimeDelay$ is fired. When $TimeDelay$ is fired, the logical time is advanced by d that is specified by the actor of $TimeDelay$. And at the output port cv, the logical time is mapped back to physical time and the output signal cv is not given until physical time $t_0 + d$. As a result, no matter how many times the composite co-simulation actor fires, the behavior of the controller in the purely functional model is preserved since it is triggered at physical time t_0 for the first time and gives the output signal at physical time $t_0 + d$.

The mapping of the logical time to physical time essentially applies implicit constraints in the contract. In this example, the $TimeDelay$ implies that $t_5 < t_0 + d$. In other words, it implies that all the end-to-end latencies of paths p_1 (from ev to cv), p_2 (from tv to cv), p_3 (from mv to cv) and p_4 (from ct to cv) must be less than d. Metronomy automatically checks these implicit constraints in the co-simulation.

In Metronomy, the user can also choose to give the output signal of a composite actor as soon as the computation is complete. If this option is chosen, the controller in the above example will give the output signal at physical time t_5. But this option will no longer guarantee the mapped model (the functional model and architectural model) keeps the same behaviors as that of the purely functional model. The functional behaviors of the mapped model depends on the performance of the architecture. Because the timing of output signal will affect the physical processes, which ultimately affects the future input and behaviors of the controller. The case study in Section 4.1 demonstrates the use of this option.

3.3 Mapping Semantics

Metronomy uses rendezvous constraints and event synchronization to implement the mapping $\mathcal{M}$ between functional and architectural models, a powerful and flexible mechanism, which allows mapping constraints to be established between arbitrary pairs of events.

Let e_f and e_a be two events in the functional and architectural models, respectively, and such that $e_a = \mathcal{M}(e_f)$. A rendezvous constraint on e_f and e_a requires that both of them be in the *proposed* state when the constraint is resolved in phase 2. Events e_f and e_a will then be set to $notified$ only when both of them are in the *proposed* state in the same round; if only one of them is *proposed*, it will be just set to $waiting$.

Particularly, as an example, if we assume e_a is proposed by the architectural model in each round, it will only be notified when the mapped event e_f is also proposed, which implies the activity in the architectural model is "driven" by the functional model. Symmetrically, if e_f is proposed by the functional model in each round, it will only be notified once e_a gets proposed, meaning that the execution of the functional model is now "driven" by the architectural model.

Table 3.1: The mapping configuration for the example in Figure 2.5

Event in the Functional Model	Event in the Architectural Model
FunctionalModel.Controller.Register2.FireBegin	Task2.Register.Begin
FunctionalModel.Controller.Register2.FireEnd	Task2.Register.End
FunctionalModel.Controller.Register.FireBegin	Task2.Register.Begin
FunctionalModel.Controller.Register.FireEnd	Task2.Register.End
FunctionalModel.Controller.AddSubtract.FireBegin	Task2.AddSubtract.Begin
FunctionalModel.Controller.AddSubtract.FireEnd	Task2.AddSubtract.End
FunctionalModel.Controller.PID.FireBegin	Task2.PID.Begin
FunctionalModel.Controller.PID.FireEnd	Task2.PID.End
FunctionalModel.Controller.Select.FireBegin	Task2.Select.Begin
FunctionalModel.Controller.Select.FireEnd	Task2.Select.End

Table 3.1 shows the mapping configuration for the example in Figure 2.5. Each actor in the controller is mapped to a service in a task by mapping the beginning and ending events of the actor to the beginning and ending events of the corresponding service. Each event is referred to by its full name in the mapping configuration.

The full name of an event in the functional model is defined as follows: {ActorFullName}.{EventType}, where {ActorFullName} includes the names of all enclosing composite actors and the name of the actor that is mapped. And {EventType} is either *FireBegin* or *FireEnd*.

The full name of an event in the architectural model is defined as follows: {ServiceName}.{EventType}, where the service name is a unique string that identifies the service and {EventType} in is either *Begin* or *End*.

Chapter 4

Case Studies

We demonstrate our methodology and the use of our co-simulation framework on design examples of embedded controllers for an aircraft electric power system and a printing press paper feed system.

4.1 Aircraft Electric Power System

4.1.1 Functional and Architectural Models with Timing Contracts

Due to the increase in electrification of modern aircraft, the design of the electrical power distribution system has become very challenging because of the safety-critical nature of the system, subject to tight reliability constraints [27].

Figure 4.1 shows a simplified functional model of a power system, including a composite *GeneratorContactorLoad* actor, modeling the power system plant, and a hierarchical controller, built out of two composite actors, the *Supervisor* and the *Controller*. The power system plant consists of a set of power sources (generators), loads and electromechanical switches (contactors), all lumped into the continuous-time actor *GeneratorContactorLoad*. We assume that the *Supervisor* is a fixed, pre-designed finite state machine which configures the power plant by actuating the contactors to connect the power sources to the loads in each aircraft operation mode.

Our goal is to explore the trade-offs involved in the design of the *Controller* that regulates and stabilizes the amplitude of the voltage on the power network, which is required to never exceed 120 V to prevent any damage in the loads. To do so, our simulation setup includes a *SingleEvent* actor modeling the insertion of a new load in the system at time 15 s.

Figure 4.2 shows the result of a purely functional simulation where the ideal controller undergoes zero end-to-end latency. The controller outputs its drive voltage (represented in blue in the figure) by instantly reacting to the voltage level sensed on the power network. The requirement is always satisfied.

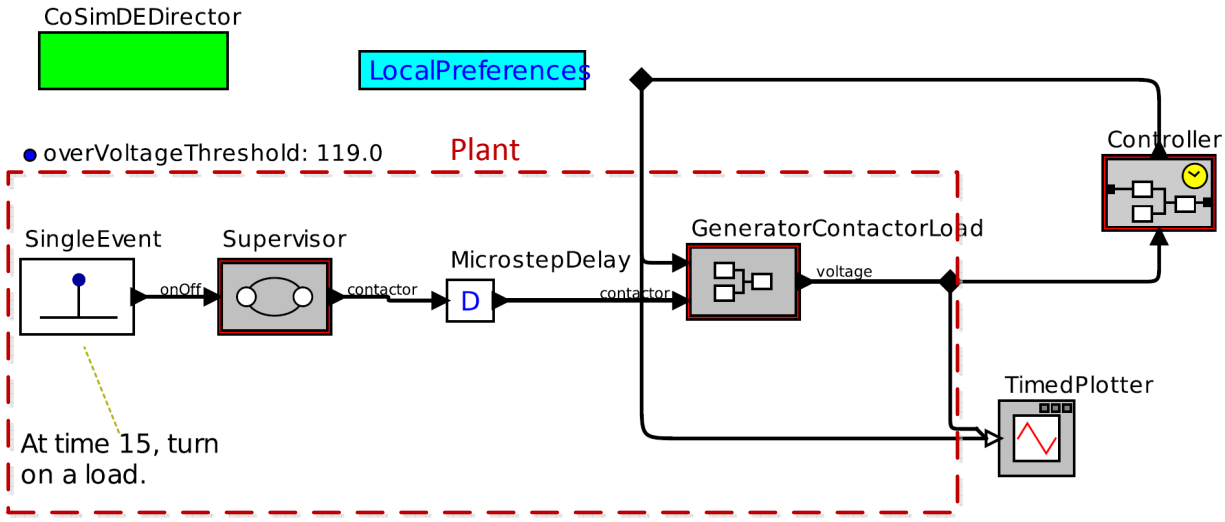

Figure 4.1: The functional model of a simplified electrical power system.

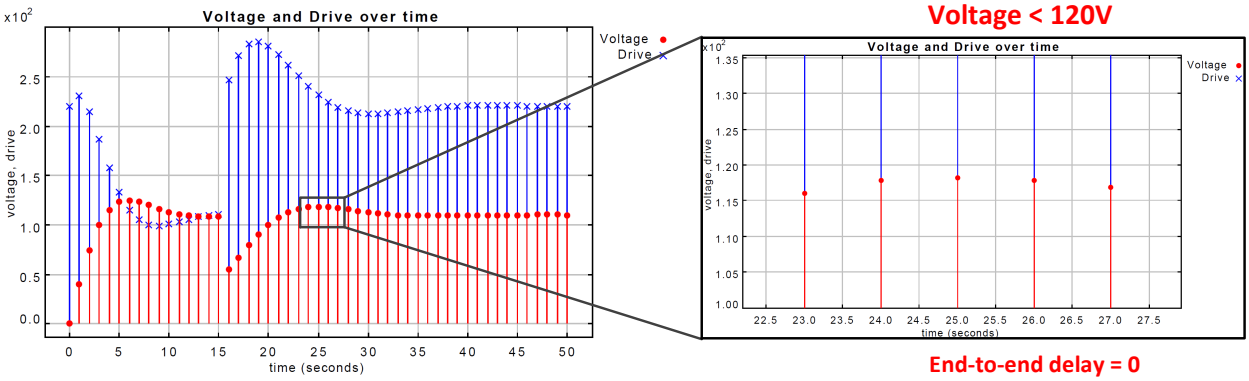

Figure 4.2: Simulation results from an ideal functional model with zero end-to-end latency.

On the other hand, in Figure 4.3, we represent a more realistic, composite *Controller* actor, where blocks *SensingComm* and *ActuatingComm* capture the effects of communication between the PID controller and the plant via sensors and actuators. Sensing and actuation delays are not always available while prototyping the control algorithm at the functional level. Therefore, instead of delving into the implementation details and the specific delay breakdown between computation and communication, the control designer may conveniently rely on a simpler interface, defined by a timing contract. In such a contract, an assumption can be made on the whole end-to-end latency between sensing and actuation, which can be captured by the following assertion:

$$t_{(ActuatingComm.e,k)} - t_{(SensingComm.b,k)} \leq d, \tag{4.1}$$

where d represents both the communication (e.g. bus) and computation (e.g. processor) delays related to the implementation architecture. Under the assumption in (4.1),

the functional model guarantees that the *Controller* is triggered every 1 second: $\forall k$, $t_{(SensingComm.b,k)} = k$. Moreover, the amount of computation in each activation is bounded. For example, PID needs to compute at most once whenever *PIDController* is triggered:

$$\begin{aligned}&\forall k, \forall j,\\&(t_{(PIDController.b,k)} \leq t_{(PIDController.PID.b,j)}\\&\wedge t_{(PIDController.PID.e,j+r-1)} \leq t_{(PIDController.e,k)}) \rightarrow (r = 1).\end{aligned}$$

The functional model is then accompanied by an architectural model, including a processor, a sensor, an actuator, a bus and a simple OS layer that supports the following services: reading values from the sensor, writing values to the actuator, arithmetic computations and PID computation. In its contract, the architecture guarantees that the delay of each service is bounded.

To provide realistic worst case estimations of the end-to-end delays, we co-simulate the composition of the functional model with the architectural model, where firing events of *SensingComm*, *ActuatingComm* and *Controller* are mapped to "service" events in the architecture. As stated earlier, such a mapping mechanism allows accounting for the impact of architectural choices on the system functionality, while keeping the details of the architecture "hidden" from the pure functional model. We perform verification of the timing contract by checking that for each event arriving at *port2* in Figure 4.3 the end-to-end timing assumption is satisfied, i.e. it is discharged by the architecture guarantees. If this is not the case, a timing violation exception will be thrown.

4.1.2 Exploring Design Choices in Architecture

As an example, we investigate the impact of the latency assumptions on the final controller design, a crucial parameter for the development of this system. A pessimistic latency bound d may end up with a degraded controller performance, while an optimistic bound at the functional level may require a fast and expensive architecture to be supported. In Figure 4.4, we prototype a controller by assuming a loose assumption on the end-to-end latency ($d = 0.3$ s). Such an assumption is compatible with an inexpensive architecture, with high (pessimistic) sensing and actuation delays, and a "slow" communication bus. However, the mapped controller violates our requirement since an overdrive voltage exceeding 120 V is observed at time 24 s and 25 s. We can then overcome this issue by either providing a "faster" bus, or modifying the functional architecture in the first place to accommodate any impairments related to the implementation platform.

The results obtained after implementing the former solution are visualized in Figure 4.5, where the functional model is now assuming $d = 0.09$ s, albeit at additional architecture-related costs.

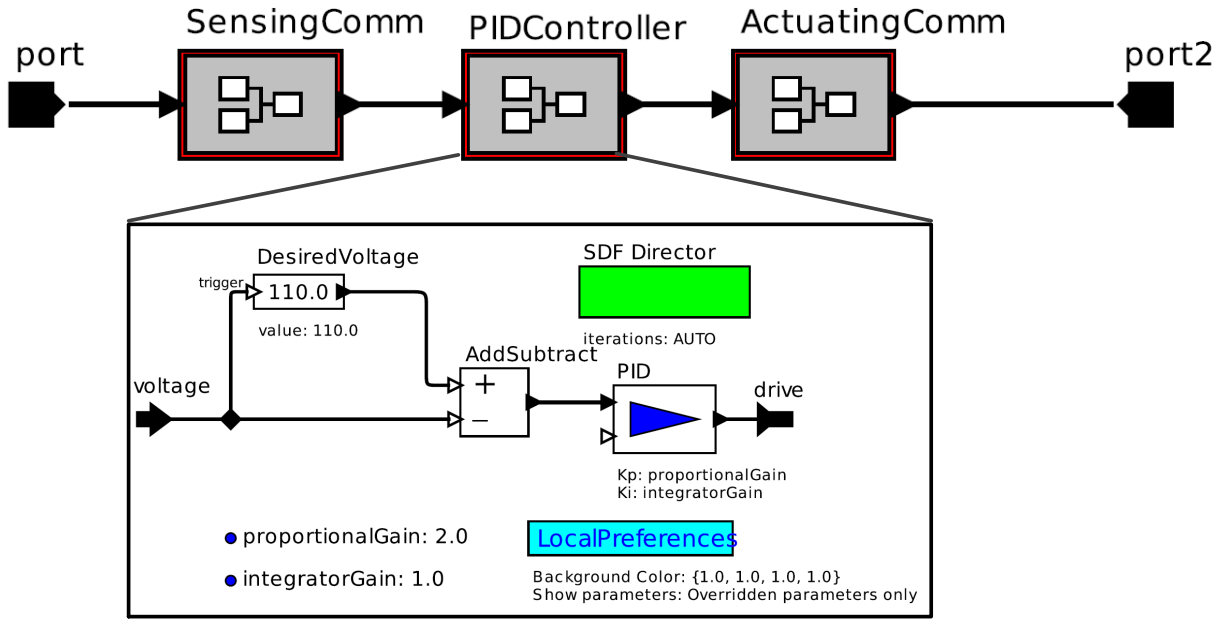

Figure 4.3: The controller in an electrical power system. *PIDController* is a sampled-data feedback controller. The *PID* control filter simply takes the difference between the measured voltage and the desired one.

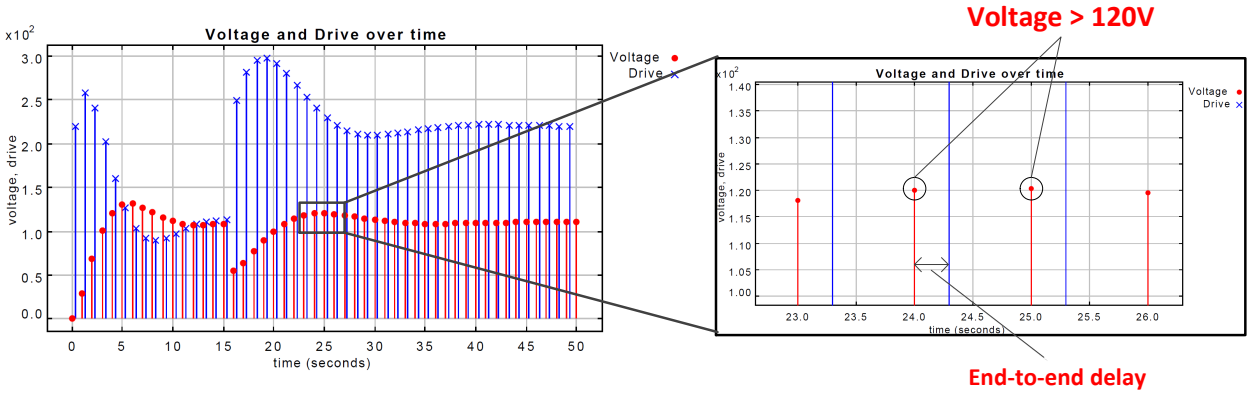

Figure 4.4: Simulation of the functional model and the architecture with a slow bus.

4.1.3 Exploring Design Choices in Function

On the other hand, Figure 4.6 shows how the functional model can be modified when the latter approach is adopted. To avoid over-voltage problems, the functional model is equipped with an additional voltage protection mechanism for the loads. Whenever the voltage level exceeds 119 V, the voltage protection kicks in and disconnects the loads from the power network until the desired voltage is restored on the line. By utilizing the additional voltage

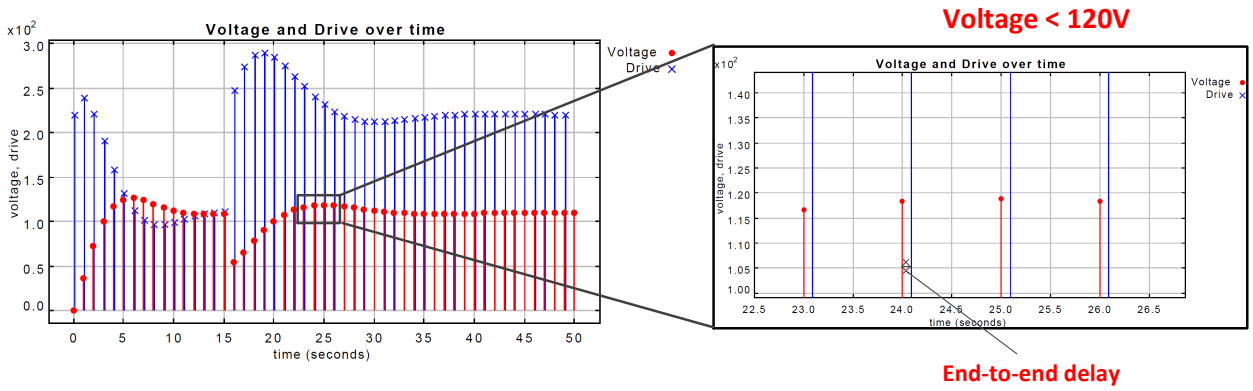

Figure 4.5: Simulation of the functional model with accelerated architecture.

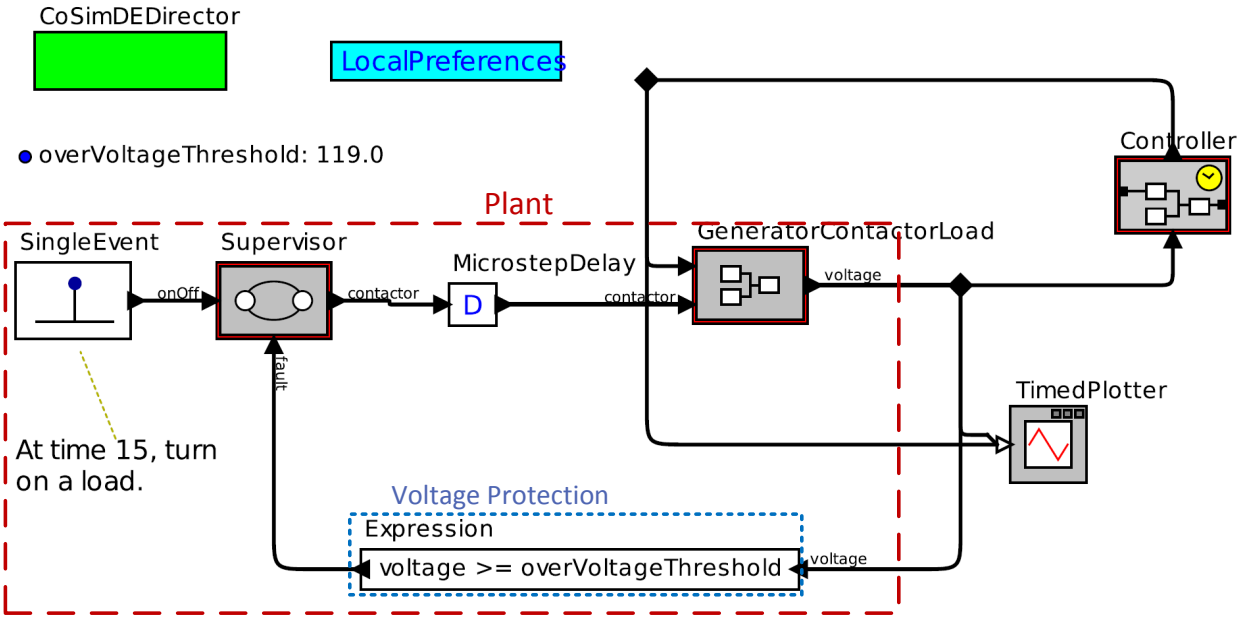

Figure 4.6: Functional model of an electrical power system with over-voltage protection.

protection, a looser bound is acceptable for the architecture contract, which allows leveraging cheaper solutions.

Figure 4.7 shows the simulation result of the mapped model with voltage protection. At time 24 s, the protection circuit detects that the voltage exceeds 119 V and disconnects the loads, which will be reconnected later on, when the voltage stabilizes. Different solutions result in different timing contracts ($d = 0.3$ s or $d = 0.09$ s) between function and architecture designers, which in turn restrict the further refinement of both function and architecture.

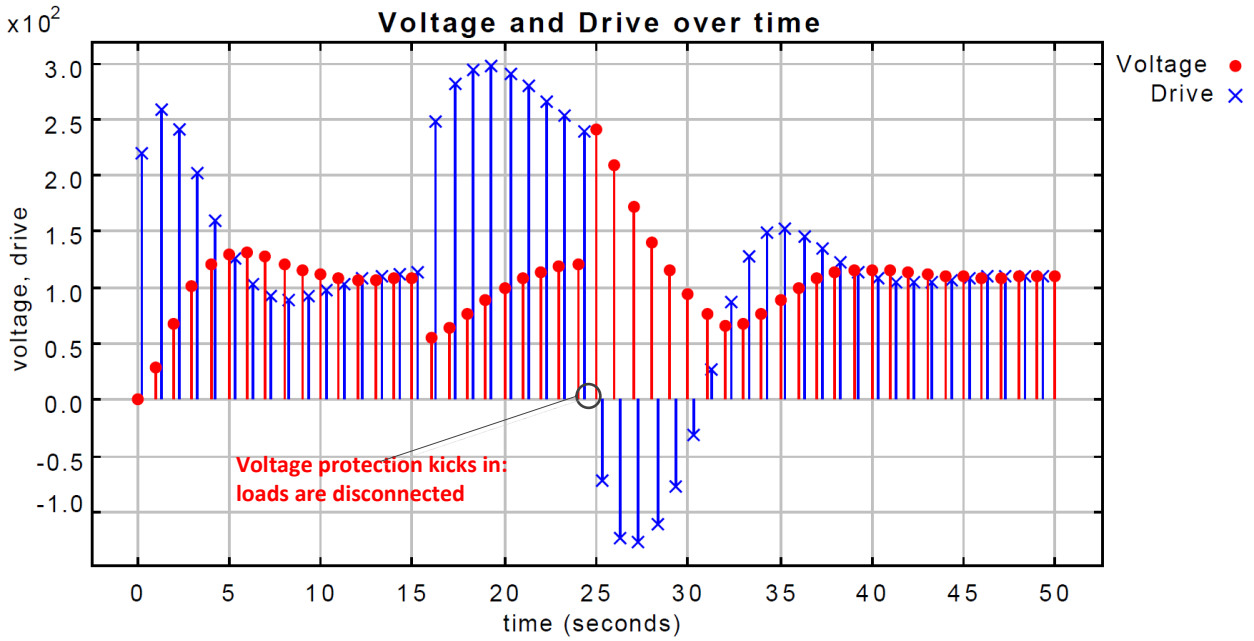

Figure 4.7: Simulation of the functional model with over-voltage protection.

4.2 Printing Press Paper Feed System

4.2.1 Functional and Architectural Models with Timing Contracts

Figure 4.8 shows the paper feed system of a high-speed printing press. The system consists of three types of rollers: two drive rollers, a feed roller, and a reserve roller. A roll of paper is driven by the feed roller to feed the printing machine. When the radius of the feed roll becomes lower than a first threshold, a signal is sent to bring up the reserve roller and its velocity will eventually match that of the drive roller. When the radius of the feed paper roll is lower than a second threshold, a tape detector begins sensing the tape on the reserve roll. When the presence of a strip of tape on the reserve roll is detected, the contact controller computes when the strip of tape will be directly opposite the contact actuator and prior to this point of time, it sends a signal to the contact actuator, which forces contact between the active and reserve paper so that they are attached by the tape. Right after that, the cutter actuator cuts the paper from the feed roller so that the reserve roll continues to feed the printing machine. The most critical scenario for timing requirements occurs when the contact actuator reacts after the radius of the feed roll falls below the second threshold and the tape is detected.

Figure 4.9 shows the functional model of the paper feed subsystem, which consists of the following modules:

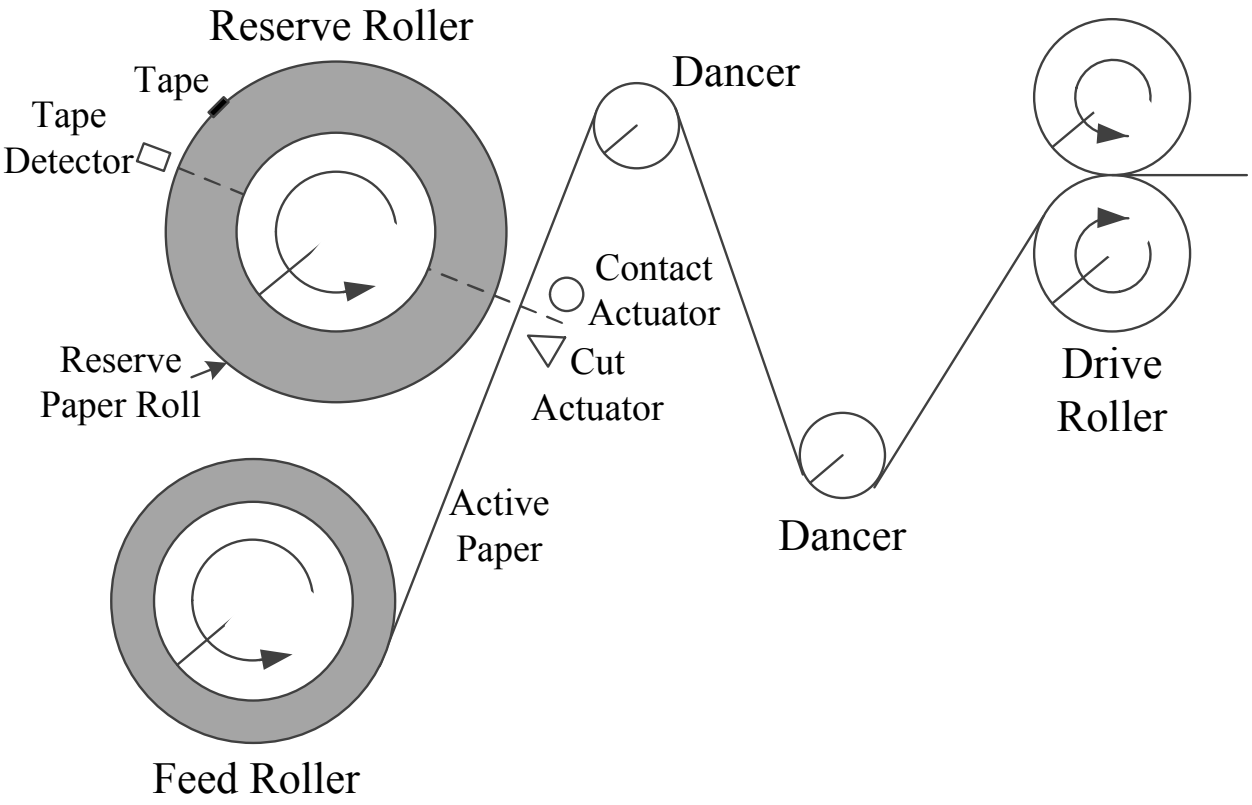

Figure 4.8: The paper feed subsystem.

- ① DriveRoller, ② FeedRoller, and ③ ReserveRoller model the paper rollers driven by motors. The motor operates in a continuous-time domain and each motor is controlled by a drive voltage. The drive voltage is a continuous-time signal. In each roller model, a hold actor in each of the roller models converts the discrete control signal into continuous-time signal by keeping the value constant until it receives a new input. And a sample actor models the sensor that measures the surface velocity of the paper roll. In addition to the surface velocity of the paper roll, on the feed roller and reserve roller, a sample actor also measures the radius of the paper roll left on the roller. The hold and sample actors interface between continuous-time plant model and discrete controller.

- ④ RemainingPaperDetector is a digital sensor that monitors the radius of the feed paper roll is lower than given thresholds.

- ⑤ TapeDetector is an analog sensor that sends a pulse signal when detecting a tape on the paper roll.

- ⑥ ContactController takes two inputs: the pulse signal (armContact) indicating the paper roll left on Feed Roller is nearly empty and the pulse signal (tapeDetector) indi-

cating the a tape is detected. When armContact is received, Contact Controller start waiting for tapeDetection. Once the tape is detected, ContactController calculates how long it would take for the strip of tape to rotate to the position that is directly opposite the contact actuator. It then schedules a timer and when the timer expires, a pulse signal is sent to Contact Actuator to make the contact so that the paper of the active roll is attached to the reserve roll. ContactController also sets another timer for the cut signal. When the timer expires, a cut signal is sent to Cut Actuator so that the paper from FeedRoller is detached. The cut signal also notifies FeedController to stop the FeedRoller.

- ⑦ DtFTrackingController (Drive to Feed Tracking Controller) takes the measured velocity of DriveRoller and the measured velocity of FeedRoller as its inputs. It computes a tracking error between the two velocities. Similarly, ⑧ DtRTrackingController (Drive to Reserve Tracking Controller) takes the measured velocity of DriveRoller and the measured velocity of ReserveRoller as its inputs and computes a tracking error between the two velocities. The tracking errors are used to adjust the target velocities in DriveController, FeedController, and ReserveController.

- ⑨ DriveController, ⑩ FeedController, and ⑪ ReserveController are feedback controllers that tries to minimize the "error" between the target velocities and the measured velocities by adjusting the drive voltage of the rollers. One output signal is generated in one sampling period ($T_{sampling}$), which updates the drive voltage at the hold actor.

- ⑫ DriveTargetVelocityProfile, ⑬ FeedTargetVelocityProfile, and ⑭ ReserveTargetVelocityProfile computes the profiled target velocities of the drive roller, the feed roller, and the reserve roller respectively.

The controller consists of nine composite actors, numbered from 6 to 14. We assume that the controller senses and actuates the plant with a sampling period $T_{sample} \in \mathcal{T}$ which is a design choice, with $\mathcal{T} = \{0.02, 0.04, 0.06, 0.08, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$ s. The architectural model *Arch* is a single processor multi-task platform. To implement the mapping function $\mathcal{M}$, each of the actors 6-14 in Figure 4.9 is mapped to a task, nine in total, which implies that all the "begin" and "end" events of the atomic actors enclosed in the composite actors are mapped to the "begin" and "end" events of the corresponding services of the tasks. These tasks are scheduled by a priority-based operating system supporting preemption. The priorities are given from high to low in the following order: 6,7,8,12,13,14,9,10,11. The processor is connected to sensors and actuators via Ethernet. End-to-end latency measurements consist of the following contributions: sensor and actuator delay, communication delay, processor execution delay. We assume that the sensing, actuation and communication delays are constants. The frequency of the processor $f_{proc} \in \mathcal{F}$ is a design choice, with $\mathcal{F} = \{3.3, 5, 10, 16.6, 20, 33.3, 50, 100, 133\}$ MHz; faster processors are more expensive.

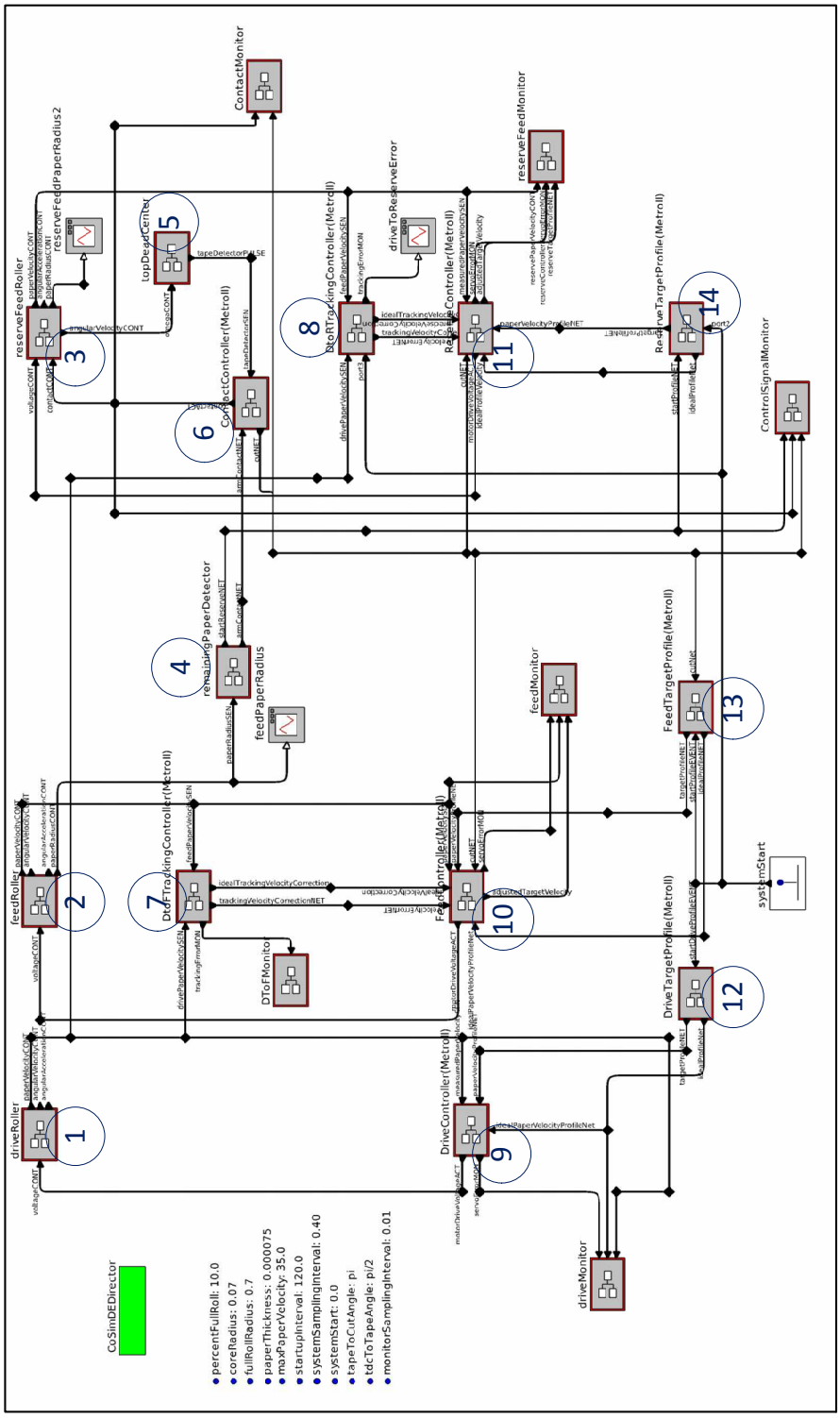

Figure 4.9: The functional model of a paper-feed subsystem: 1. Drive Roller; 2. Feed Roller; 3. Reserve Roller; 4. Remaining Paper Detector; 5. Tape Detector; 6. Contact Controller; 7. Drive to Feed Tracking Controller; 8. Drive to Reserve Tracking Controller; 9. Drive Controller; 10. Feed Controller; 11. Reserve Controller; 12. Drive Target Velocity Profile; 13. Feed Target Velocity Profile; 14. Reserve Target Velocity Profile; The rest are monitors.

A key performance metric in this system is the tracking error between the paper velocity of the feed (reserve) roller and the one of the drive roller. Although the two dancers shown in Figure 4.8 help compensate the difference in the velocities of the drive roller and the feed roller, a controller is still needed to minimize the tracking error, especially when the drive roller accelerates soon after the system starts. Since the radius of paper roll is also changing, the drive signal to the feed roller has to be dynamically adjusted to maintain a proper surface velocity. We add monitors to the functional model to measure the RMS tracking error ϵ_{RMS} using the following formula:

$$\epsilon_{RMS} = \sqrt{\frac{T_{mon}}{T_{sim}} \sum_{i=0}^{T_{sim}/T_{mon}} \left(V_{drive}^{iT_{mon}} - V_{feed}^{iT_{mon}}\right)^2},$$

where T_{mon} is the sampling period of the monitor, V_{drive}^{t} and V_{feed}^{t} are the velocities of the drive roller and the feed roller at time t respectively, and T_{sim} is the duration of the simulation.

We cast the design space exploration problem as a multi-objective optimization problem subject to timing contracts; we aim to minimize:

$$\begin{aligned} &\min_{T_{sample}, f_{proc}} \quad (\epsilon_{RMS}, f_{proc}) \\ &\text{s.t.} \begin{cases} Func(T_{sample}) \times Arch(f_{proc})|_{\mathcal{M}} \\ \models (\mathcal{A}_f, \mathcal{G}_f(T_{sample})) \otimes (True, \mathcal{G}_a(f_{proc})) \\ Func(T_{sample}) \times Arch(f_{proc})|_{\mathcal{M}} \models \mathcal{C}_s \\ T_{sample} \in \mathcal{T}, f_{proc} \in \mathcal{F} \end{cases} \end{aligned} \tag{4.2}$$

where both the functional and architectural contracts have been concisely denoted as pairs of assumptions and guarantees, by dropping the set of events and time tags. $\mathcal{A}_f$, $\mathcal{G}_f(T_{sample})$ and $\mathcal{G}_a(f_{proc})$ are obtained by composition of the contracts of all the controllers, among which the contract for the *Feed Controller* has been illustrated as an example in Section 2.3.2. $\mathcal{C}_s$ is the contract that specifies the system level requirements (e.g. the timing requirements on the contact actuator and the cut actuator).

4.2.2 Exploring Design Choices in Architecture

The tracking error ϵ_{RMS} depends on the sampling period T_{sample} and the end-to-end latency l of the feedback controller ($l \leq T_{sample}$). As shown in Figure 4.10, as the sampling period increases, the tracking error (in blue) significantly increases. In addition, the velocity of the paper roll becomes unstable when $T_{sample} \geq 0.6$. Figure 4.13 shows the linear velocity of the feed roller, the target velocity, and the error between them when the sampling rate is 0.8 s, which causes the system to fail.

In Figure 4.10, the curve in red shows the tracking error ϵ_{RMS} versus f_{proc} Pareto front. Given a processor speed f_{proc} we can find the sampling period T_{sample} that minimizes the tracking error while satisfying all the timing constraints. For example, when $f_{proc} = 10$ MHz,

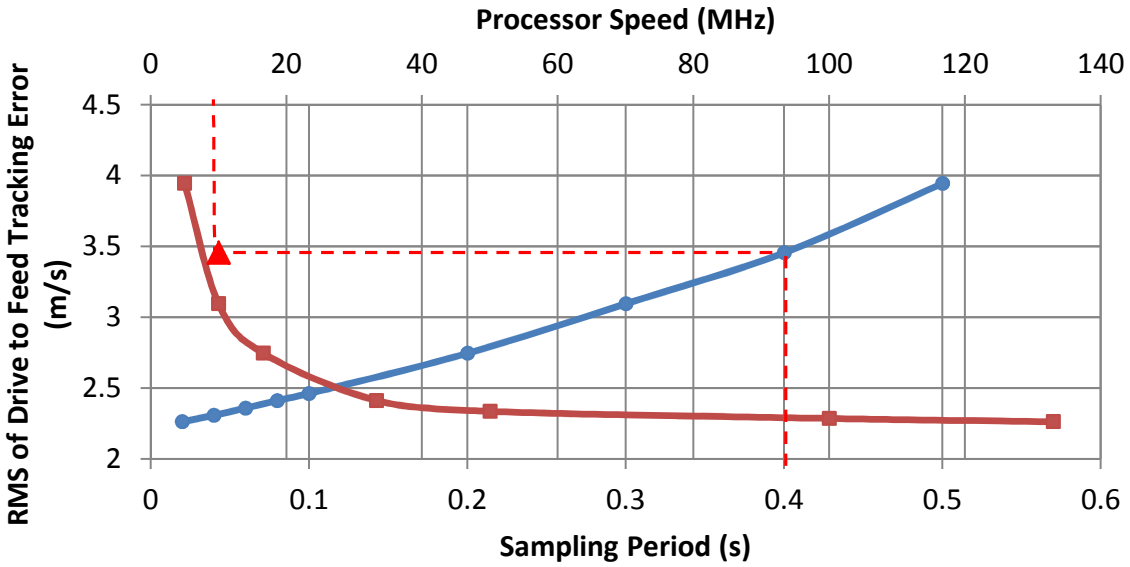

Figure 4.10: Design space exploration results, while minimizing both the tracking error and the processor speed.

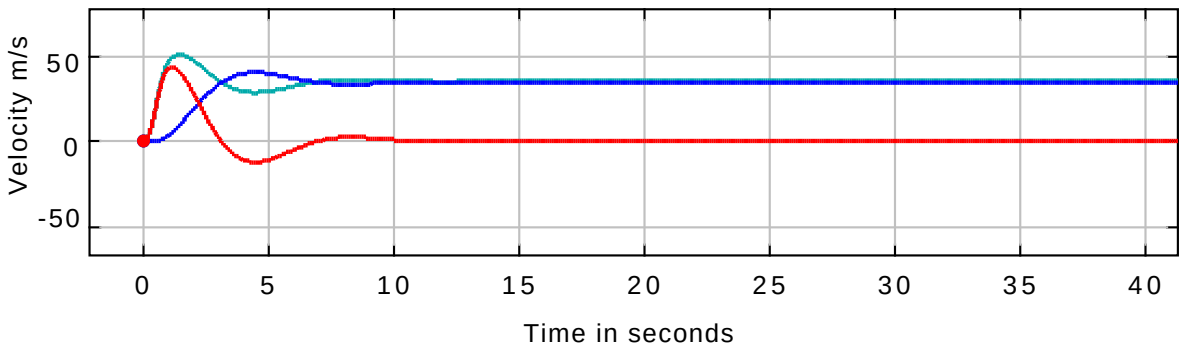

Figure 4.11: Simulation results for the paper feed system, including the target velocity (green) of the feed roller, the actual velocity (blue), and the error (red) when $T_{sample} = 0.1$ s, $f_{proc} = 33$ MHz.

the optimal ϵ_{RMS} is 3.1 m/s, as obtained from the red curve in Figure 4.10. This corresponds to an optimal sampling period of $T_{sample} = 0.3$ s, as obtained from the blue curve in Figure 4.10. Any point below the ϵ_{RMS}-versus-f_{proc} Pareto front is an infeasible design due to timing violations (e.g. $f_{proc} = 10$ MHz, $T_{sample} = 0.2$ s). Any point above the Pareto front is instead non-optimal; e.g. $T_{sample} = 0.4$ s is not an optimal choice for a 10-MHz frequency since the point (10, 3.46) is dominated by (10, 3.1), obtained for $T_{sample} = 0.3$ s.

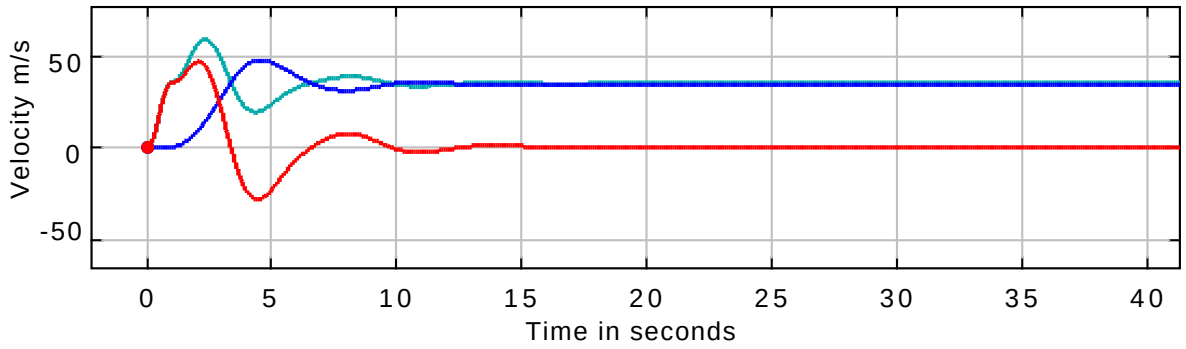

Figure 4.12: Simulation results for the paper feed system, including the target velocity (green) of the feed roller, the actual velocity (blue), and the error (red) when $T_{sample} = 0.5$ s, $f_{proc} = 5$ MHz.

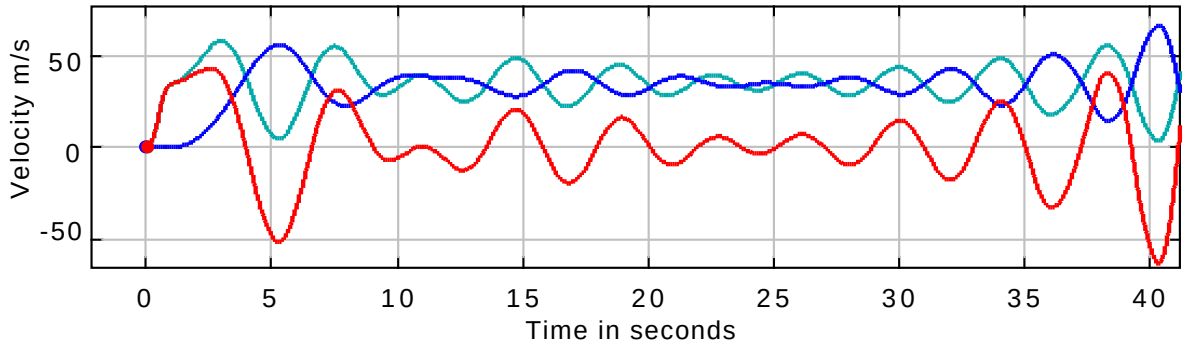

Figure 4.13: Simulation results for the paper feed system, including the target velocity (green) of the feed roller, the actual velocity (blue), and the error (red) when $T_{sample} = 0.8$ s, $f_{proc} = 3.3$ MHz.

4.2.3 Exploring Design Choices in Function

Given an architecture, we can also explore the design space at the functional level. For example, Figure 4.14 shows that with an economic processor (5 MHz) and a slow sampling rate (0.5 s) it is still possible to satisfy our tracking error requirement, but at the cost of a smaller roll acceleration. The compromise at the functional level enables us to use inexpensive platforms. Once that sampling period and the processor speed are decided, the functional and architectural timing contracts are also defined, and can be used for the next steps in the design process. Therefore, while the two models are kept separated, their interaction is still captured by the timing contract, which, together with co-simulation, makes it easier to verify the impact of different choices and explore trade-offs across the function/architecture

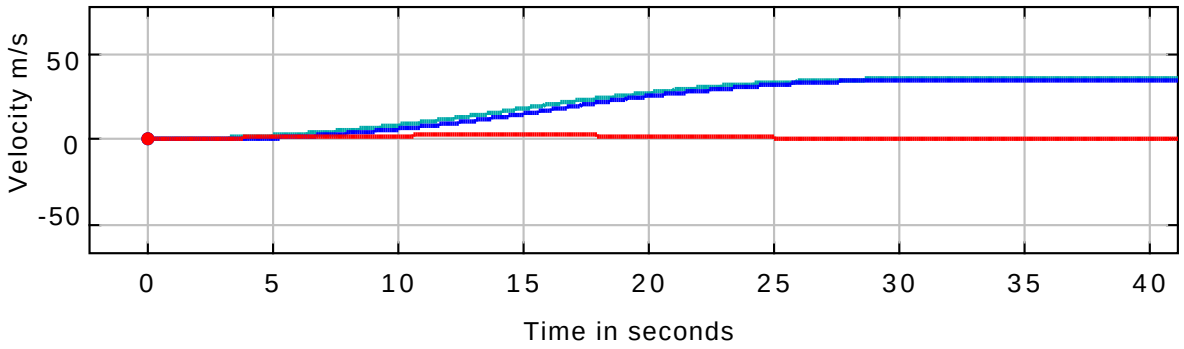

Figure 4.14: Simulation results for the paper feed system, including the target velocity (green) of the feed roller, the actual velocity (blue), and the error (red) when $T_{sample} = 0.5$ s, $f_{proc} = 5$ MHz, but with a much slower ramp-up speed.

boundary.

Chapter 5

Conclusions and Future Work

5.1 Closing Remarks

We have proposed a methodology for the verification and design space exploration of cyber-physical systems subject to real time constraints. In our framework, co-simulation of a high-level, functional model together with a lower-level, architectural model is used to *accurately* capture the effects of the implementation platform and the physical plant on the system functionality. Assumption-guarantee contracts are used to *rigorously* formalize the timing requirements at different levels of abstractions and generate simulation monitors.

To support our methodology, we have implemented Metronomy, a *versatile* co-simulation framework that enables the integration of the most suitable modeling environments to capture both the functional and architectural aspects of a design. In Metronomy, the functional aspect is captured by exploiting the *variety* of models of computation made available by the Ptolemy environment and its intuitive graphical user interface. The architectural aspect is captured within the MetroII environment, capable of modeling implementation platforms to a greater level of *detail*. Models in the two environments are co-simulated based on a rigorous *mapping* semantics. The effectiveness of our approach is demonstrated on the design of embedded controllers for an aircraft electrical power system and a paper-feed sub-system of a high-speed printing press.

5.2 Future Work

The future work mainly includes standardizing the interactions between the functional model and the architectural model as well as the mapping and the timing contract between them. In recent years, Functional Mock-up Interface (FMI) [12] has become an industrial standard to support model exchange and co-simulation of dynamic models via a combination of xml-files and compiled C-code. We believe that Metronomy can be further extended to support function-architecture co-simulation using FMI, which would open up new possibilities of integrating models from other tools. And the extension would also contribute back to FMI

standard and ultimately provides a function-architecture co-simulation standard with focus on timing verification and design space exploration, where the contract theory would play a more important role.

Bibliography

[1] Perry Alexander. *System Level Design with Rosetta*. Elsevier, 2006.

[2] Luca de Alfaro and Thomas A. Henzinger. "Interface Automata". In: *Proceedings of European Software Engineering Conference*. ESEC/FSE-9. 2001, pp. 109–120.

[3] A. Bakshi and A. Ledeczi. "MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems". In: *ACM SIGPLAN Notices*. 2001, pp. 82–93.

[4] F. Balarin et al. "Metropolis: an integrated electronic system design environment". In: *Computer* 36.4 (2003), pp. 45–52.

[5] Ananda Basu, Marius Bozga, and Joseph Sifakis. "Modeling Heterogeneous Real-time Components in BIP". In: *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*. SEFM '06. 2006, pp. 3–12.

[6] Albert Benveniste et al. "Formal Methods for Components and Objects". In: ed. by Frank S. Boer et al. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. Multiple Viewpoint Contract-Based Specification and Design, pp. 200–225.

[7] Christopher Brooks et al., eds. *Heterogeneous concurrent modeling and design in Java (Volume 1: Introduction to Ptolemy II)*. Tech. rep. UCB/ERL M05/21, University of California, Berkeley, 2005.

[8] D. Cancila et al. "Toward Correctness in the Specification and Handling of Non-Functional Attributes of High-Integrity Real-Time Embedded Systems". In: *IEEE Transactions on Industrial Informatics* 6.2 (May 2010), pp. 181–194.

[9] J. Castrillon, R. Leupers, and G. Ascheid. "MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs". In: *IEEE Transactions on Industrial Informatics* 9.1 (2013), pp. 527–545.

[10] Abhijit Davare et al. "MetroII: A design environment for cyber-physical systems". In: *ACM Transactions on Embedded Computing Systems* 12.1s (2013), 49:1–49:31.

[11] Patricia Derler et al. "Cyber-physical system design contracts". In: *Proc. International Conference on Cyber-Physical Systems*. Philadelphia, Pennsylvania, 2013, pp. 109–118.

[12] *Functional Mock-up Interface @ONLINE*. URL: `http://www.fmi-standard.org/`.

[13] Arkadeb Ghosal et al. "A hierarchical coordination language for interacting real-time tasks". In: *Proc. International Conference on Embedded Software*. 2006, pp. 132–141.

[14] Thorsten Grötker et al. *System Design with SystemC*. Springer, 2002.

[15] Liangpeng Guo et al. "Metronomy: A Function-architecture Co-simulation Framework for Timing Verification of Cyber-physical Systems". In: *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. CODES '14. ACM, 2014, 24:1–24:10.

[16] T.A. Henzinger et al. "From control models to real-time code using Giotto". In: *IEEE Control Systems* 23.1 (2003), pp. 50–64.

[17] Axel Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers Inc., 2003.

[18] Gilles Kahn. "The Semantics of a Simple Language for Parallel Programming". In: *IFIP Congress* (1974).

[19] G. Karsai et al. "Model-integrated development of embedded software". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 145–164.

[20] Hokeun Kim et al. "A tool integration approach for architectural exploration of aircraft electric power systems". In: *Proceedings of International Conference on Cyber-Physical Systems, Networks, and Applications*. 2013, pp. 38–43.

[21] Akos Ledeczi et al. "Modeling Methodology for Integrated Simulation of Embedded Systems". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 13.1 (Jan. 2003), pp. 82–103.

[22] Akos Ledeczi et al. "The generic modeling environment". In: *Workshop on Intelligent Signal Processing, Budapest, Hungary*. Vol. 17. 2001.

[23] Paul Lieverse, Pieter van der Wolf, and Ed Deprettere. "A trace transformation technique for communication refinement". In: *Proceedings of International Symposium on Hardware/Software Codesign*. 2001, pp. 134–139.

[24] Jie Liu et al. "Actor-oriented control system design: a responsible framework perspective". In: *IEEE Transactions on Control Systems Technology* 12.2 (Mar. 2004), pp. 250–262.

[25] Sandeep Neema, Janos Sztipanovits, and Gabor Karsai. "Constraint-based design-space exploration and model synthesis". In: *Proceedings of International Conference on Embedded Software, EMSOFT'03, Volume 2855 of LNCS*. Springer, 2003, pp. 290–305.

[26] H. Nikolov et al. "Daedalus: toward composable multimedia MP-SoC design". In: *Proc. Design Automation Conference*. 2008, pp. 574–579.

[27] P. Nuzzo et al. "A Contract-Based Methodology for Aircraft Electric Power System Design". In: *IEEE Access* 2 (2014), pp. 1–25.

[28] H.D. Patel, S.K. Shukla, and R. Bergamaschi. "Heterogeneous Behavioral Hierarchy Extensions for SystemC". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 26.4 (Apr. 2007), pp. 765–780.

[29] A.D. Pimentel, C. Erbas, and S. Polstra. "A systematic approach to exploring embedded system architectures at multiple abstraction levels". In: *IEEE Transactions on Computers* 55.2 (2006), pp. 99–112.

[30] Claudius Ptolemaeus, ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL: `http://ptolemy.org/books/Systems`.

[31] I Sander and A Jantsch. "System modeling and transformational design refinement in ForSyDe [formal system design]". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.1 (Jan. 2004), pp. 17–32.

[32] Alberto Sangiovanni-Vincentelli. "Defining Platform-Based Design". In: *EEdesign* (2002).

[33] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems". In: *European Journal of Control* (2012).

[34] The Metropolis Project Team. *The Metropolis Meta Model Version 0.4*. September 14, 2004.

[35] Yang Zhao, Jie Liu, and Edward A. Lee. "A Programming Model for Time-Synchronized Distributed Real-Time Systems". In: *Proceedings of Real Time and Embedded Technology and Applications Symposium*. 2007, pp. 259–268.